UNIVERSIDAD DE BELGRANO

Las tesinas de Belgrano

Facultad de Tecnología Informática
Carrera Ingeniería Informática

Proyecto TEGE: Desarrollo de un juego de computadora

N° 343 Martín Gabriel Opromolla

Tutor: Lic. Martín D. Cernadas
Director de carrera: Ing. Víctor Rodríguez

Departamento de Investigaciones
Año 2008

Universidad de Belgrano
Zabala 1837 (C1426DQ6)
Ciudad Autónoma de Buenos Aires - Argentina
Tel.: 011-4788-5400 int. 2533
e-mail: invest@ub.edu.ar
url: http://www.ub.edu.ar/investigaciones

Resumen

En el universo del software existe un segmento atrapante por demás, como es el de los juegos. Sumo a ello mi pasión por los mismos, aunando mis conocimientos técnicos en pos de explorar y producir un producto que no solo sea un software más que motive un caso de estudio y evaluación, sino que sea la síntesis por la cual puedo expresarme como un futuro profesional que logró aunir sus deseos con su trabajo.

Dentro de los segmentos de juegos, enfocaré mi trabajo en los juegos de estrategia para elaborar este producto. Y dentro del software, en la programación orientada a objetos, multiplataforma, enfocado en el desarrollador mas que en el usuario. Comercialmente, en un análisis cruzado de consolas y computadoras disponibles, con comparativas de costos, performance y disponibilidad con facilidad de programación.

El producto de software que es un juego completo (no prototipo o demo) del género de estrategia por turnos (TBS, turn-based strategy). El mismo estará basado en el juego de mesa llamado T.E.G. (plan Táctico y Estratégico de la Guerra) o Risk (versión americana). El juego les da la posibilidad a los jugadores de conquistar un mapa del universo utilizando ejércitos que, en este caso, son naves espaciales para dicho fin. Las reglas básicas disponen que los planetas de un jugador puedan atacar a los planetas de otro jugador. La obtención de nuevos planetas es premiada con nuevas naves o ejércitos que le ayudaran al jugador a cumplir con el objetivo final que es la conquista de este universo. En el juego de mesa el combate se hace mediante dados en donde el jugador que ataca y el que defiende tiran algunos dados. Aquel que consiga los números más altos en los dados va eliminando los ejércitos del jugador contrario. Si el país defensor se queda sin esos ejércitos, el jugador atacante tomará posesión del territorio.

El producto de software contará con las reglas del juego y una jugabilidad similar aunque distinto escenario y por ende tablero. El juego estará creado con herramientas de Microsoft, con el lenguaje C# y la librería XNA Framework.

Índice

1. Introducción

En la actualidad es imposible no encontrarse con eventos dedicados a los videojuegos, conocer a alguien que tenga una consola, escuchar una conversación sobre algún juego o incluso uno mismo dedicar algunos minutos al día en alguna pagina de Internet con uno de los tantos juegos casuales que existen como para distraerse un momento.

La industria de los videojuegos fue evolucionando con el tiempo, no sólo tecnológicamente sino también entendiendo al consumidor, entendiendo que no hay que llegar solamente al “hardcore” gamer o sea al jugar habitual, el que más compra y mas activo. Las empresas descubrieron así al “casual” gamer aquél que también quiere jugar, aunque sea en sesiones cortas y con juegos más simples que generalmente no son los comerciales que se distribuyen a nivel global. Ese nicho del mercado, casi inexplorado hace 10 años ahora es tomado muy en serio, apuntando a personas mayores, adultas y a las mujeres de todas las edades con los juegos denominados “casual” games, con algunos ejemplares en PC pero mas centralizados en juegos Web y para celulares.

Este crecimiento por parte de la industria vino acompañado por cambios sociales. En los 70’ y 80’ los juegos eran una actividad para los que se consideraban rebeldes, adolescentes tratando de expresar su libertad que eran mirados de forma despectiva por el resto de la sociedad. Eso fue cambiando cuando las personas se dieron cuenta de que lo que estaban haciendo estos “rebeldes” no era algo malo y de que ellos mismo tenían ganas de satisfacer sus necesidades lúdicas.

1.2. Objetivo

Desarrollar un juego completo y complejo, o sea no casual, utilizando la librería de Microsoft XNA Framework y el lenguaje C#.

1.3. Justificación

La elección del tema sobre el desarrollo de un software surgió por gustos personales en la programación y se entendió que era un ejercicio válido y practico que integraba varios conceptos aprendidos durante la carrera. En particular la idea del desarrollo de un juego nació de la sugerencia de que la tesina podía estar asociada a un hobbie, lo que en este caso se traduce claramente a la programación de juegos. La idea se tomó como buena porque si la actividad era parte de un hobbie, el esfuerzo y dedicación hacia la tesina serán mayores. Igualmente en esta tesina hay otro fondo que va más allá del cumplimiento académico. Esta

tesina es importante para mí ya que me va ayudar en lo que espero se convierta en mi futuro laboral, la experiencia ganada en la programación de un lenguaje clave como C# y herramientas como DirectX serán invaluables.

Se considera como aporte a todo el proyecto en sí. Un proyecto de este estilo unifica todos los conceptos aprendidos durante la carrera y demuestra las capacidades personales y profesionales para concretar un proyecto de software complejo, poniendo énfasis en la disciplina de programación fundamentalmente y otras relacionadas al proceso de software como Diseño del mismo.

Además la industria de los videojuegos es una industria creciente, una industria que cada vez mueve más dinero y de la que vale la pena ser parte.

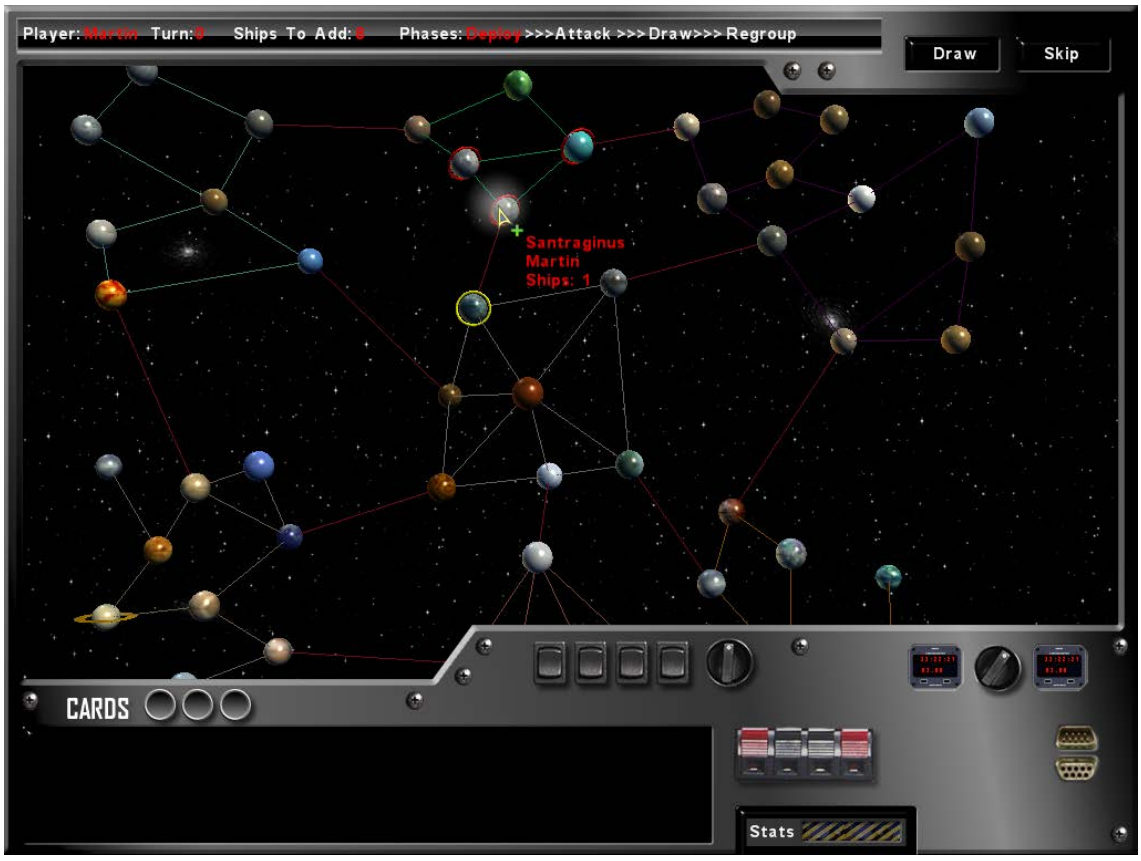
1.4. Hipótesis

Es posible desarrollar un juego comercial, sin gasto de licencias, solo con el costo de horas/hombre de programación. Llegar a comercializarlo o hacerlo conocido para tener la oportunidad de que un comercial lo vea y lo venda por nosotros, ya sea como aplicación independiente para WINDOWS, X-BOX o incluso unida a plataformas WEB 3 (como Facebook está orientado), es un hecho hoy en día.

1.5. Descripción Funcional

- El juego permitirá a dos jugadores jugar en modo de juego “Hot Seat” (ambos jugadores jugando desde la misma computadora) entre ellos.
- El jugador interactúa con el juego para ganarle al otro jugador.
- Posee elementos de multimedia como gráficos 2D, gráficos 3D y sonidos.
- En el menú se pueden editar las opciones de volumen para los sonidos.

A continuación, una imagen del juego:



1.6. Límites

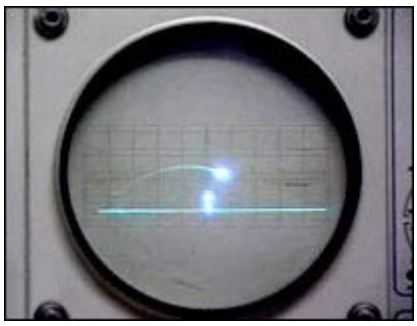
- El juego esta pensado para plataforma Windows solamente. Aunque por las características de C# y de XNA puede ser portado a la consola XBOX360, aunque no está incluido como objetivo de este proyecto en esta etapa.
- No soportará multi-jugador en red. La primera versión del XNA no lo soportaba, luego la versión 2.0 si incluyó herramientas para comunicación TCP/IP pero el proyecto ya esta diseñado sin esa característica. Los jugadores juegan desde una sola computadora.
- No tendrá seguridad de ningún tipo salvo la provista por el XNA para el cuidado del contenido multimedia y del código.
- El juego esta hecho para jugadores humanos por ende el juego no posee ninguna programación relacionada con IA (inteligencia artificial).
- Por el tipo de videojuego no existen conceptos de física aplicada.

2. Historia y Contexto

A continuación paso a relatar los eventos que sucedieron al mismo tiempo que los primeros clásicos de los videojuegos surgieron. Esta primera etapa se la conoce como la Golden Age (época dorada) de los videojuegos, en donde aparecieron los primero prototipos, se creó la industria y donde la creación de juegos estaba estipulada por la tecnología con la que se disponía.

2.1. Época Dorada

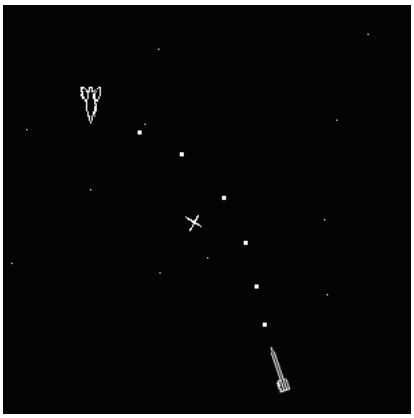
La acción de apretar un botón en la actualidad no representa nada especial ya que es parte de nuestras vidas cotidianas en celulares, electrodomésticos, cualquier aparato en general. Los juegos tienen un comienzo oscuro, ya que hace 50 años el hecho de apretar un botón significaba ganar una guerra. Aunque con propósitos diferentes, la tecnología de ese entonces trataba de simular "juegos" de guerra para conocer los resultados posibles de un combate o guerra total. La guerra fría se trato de simulaciones compitiendo entre sí por parte de Estados Unidos y de la actual Rusia. Por suerte ese concepto fue cambiando. Se considera al primer juego el que en 1958 William Higinbotham (que participó del proyecto para la bomba atómica) desarrolló llamado "Tennis for Two" (tenis para dos) en un osciloscopio, por supuesto este juego nunca fue comercial, sólo demostró que la tecnología que se usaba en el momento, con creatividad, podía ser usada para fines más interesantes que la guerra.



Tennis For Two

A comienzos de los 60' el mundo comenzó su carrera espacial, Estados Unidos y la Unión Soviética enfrentados otra vez por ver quien mandaba la primer persona al espacio. En 1962 Steve Russell que pertenece al MIT creó el juego "Spacewar!" (Guerra del espacio), influenciado por lo que ocurría en el mundo y la paranoia que se estaba gestionando de una guerra espacial. Este juego ponía a dos jugadores enfrentados, cada uno con una nave espacial y tratando de eliminar la nave del otro jugador. Este juego puso los fundamentos para una de las piezas más importantes dentro de los videojuegos, el joystick.

El juego funcionaba sobre la PDP-1 mini-computadora, no se hizo comercial pero se distribuyó al estilo de open-source para que cualquier Universidad con una PDP-1 lo pudiese tener.



SpaceWar!

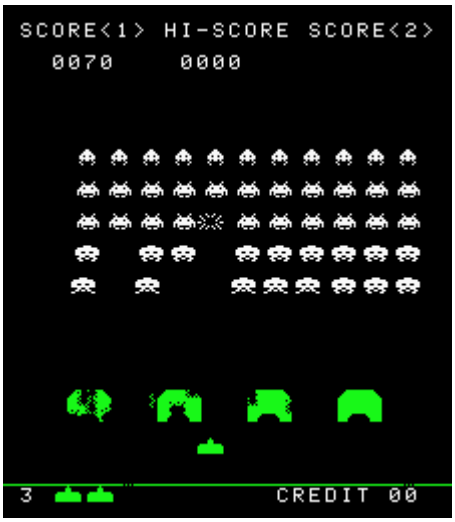
Durante los 60' y comienzos de los 70' el mundo estuvo en guerra, la guerra de Vietnam. Todo lo que se veía en la televisión eran reportes de la guerra. La audiencia necesitaba un escape de esa realidad. Los primero juegos reflejaron conflictos del momento pero en 1968 Ralph Baer pensó que la televisión podía ser usada para otra cosa que sólo malas noticias. Considerado el "Padre" de los videojuegos creo la primer consola llamada Maganavox Odyssey que venía con 7 juegos que eran variaciones del "Tennis for Two" y del que luego sería conocido como Pong.

Magnavox Odyssey

Hasta ahora se vieron prototipos de juegos y consolas que no formaban un mercado, simplemente existían. En 1972 Nolan Bushnell, considerado el padre de la Industria de los videojuegos, fundó la empresa Atari y sacó uno de los juegos más conocidos, el Pong. Este juego que se encontraba en bares y negocios, permitía a personas que no se conocían competir entre sí para demostrar quien era el mejor. Tenía una lógica más "compleja" que la que se encontraba en la Magnavox Odyssey y tenía los primeros efectos de sonido. En la misma época las mujeres peleaban por sus derechos de igualdad y encontraron en el Pong que con la igualdad de condiciones que ofrecía el juego más una mejor coordinación en los músculos pequeños, en general, eran mejores jugadoras que los hombres, llegando a desafiarlos, apostando y ganando dinero.

Pong

Desde la segunda guerra mundial Japón creció tecnológicamente de forma exponencial. Alentando la educación en el área de tecnología la juventud japonesa estaba muy bien formada en esta área. Viendo los progresos tecnológicos del mundo Japón tomó los mismos, los mejoró y los produjo masivamente haciéndolos más baratos. Ya estando en la cima tecnológica, en 1978 Tomohiro Nishikado puso a Japón a la vanguardia de la industria de los videojuegos también. Creó el "Space Invaders", juego que agregó dos conceptos claves, música y que a su vez esa música tenía tempo variable atrapando al jugador aún mas en el juego. A partir de este momento se produjo una revolución social, a la noche los dueños de los negocios sacaban su mercadería habitual para poner en sus locales maquinas del "Space Invaders". Las familias no veían con buenos ojos que sus hijos saliesen a la noche para meterse en estos locales y los jóvenes llegaban a robar dinero de los padres o a hacer monedas falsas para poder jugar. Fue la primera vez que hubo un quiebre generacional, la gente adulta simplemente nunca llego a entender como un juego lograba ese efecto.

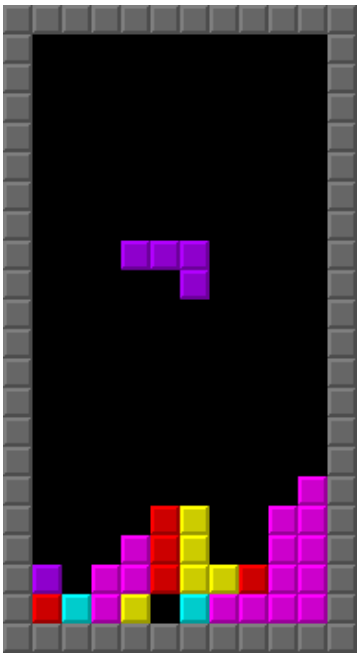


Space Invaders

Los juegos hasta este momento tenían un fondo casi pesimista porque, como lo dictaba la histeria popular del momento por las guerras, eventualmente el jugador terminaba perdiendo, también eran violentos y las ideas rondaban en disparar y destruir. En 1980 Toru Iwatani tratando de llevar los videojuegos hacia otro rumbo y también tratando de llegar al mercado femenino no usando juegos violentos creo el “Pac-Man”, la primera estrella de los videojuegos. Un juego con muchos colores, sonidos, música y niveles que el usuario tenía que superar dándole al usuario esa satisfacción de victoria que faltaba hasta ese momento. Era la primera vez que existía el concepto de protagonista.

Pac-Man

En 1985 en una Rusia con casi sin entretenimiento, Alexey Pajitnov, que trabajaba en la Academia Rusa de Ciencia como Ingeniero en Computadoras creó el “Tetris”. El videojuego considerado más adictivo de la historia que también fue el más portado a consolas y computadoras y con más versiones conocidas. Era la primera vez que se obligaba al jugador pensar en tiempo real para solucionar la mecánica del juego. El Tetris es el segundo videojuego más vendido de la historia con 33.000.000 de ventas. Por la Unión Soviética comunista el gobierno tomó su juego y tomando como modelo a Atari lo comercializó ganando millones con él. Recién en 1996 Alexey recuperaría los derechos sobre su Tetris.



Tetris

2.2. Evolución

A comienzos de los 80’ hubo una crisis de los videojuegos, Atari que en ese momento fue vendida a Warner creó la Atari 2600, la primer consola que soportaba distintos juegos no sólo un conjunto de juegos iniciales. Aunque tuvo éxito al principio Warner cambio la metodología de trabajo que tenía Atari, la hizo menos informal y la convirtió en una empresa dedicada sólo a hacer dinero. De esta forma los juegos fueron cada vez peor al punto que Atari quebró y muchos de los juegos fueron directamente enterrados y cubiertos de cemento (como cuenta la historia del juego de E.T. que es considerado el peor de la historia). Atari era la única empresa en esta industria así que con esta crisis por supuesto se abrieron las puertas para que compañías menores interesadas en incursionar en el área con ideas nuevas tuvieran su oportunidad de hacerlo.

Los videojuegos tuvieron que evolucionar, luego del estancamiento sufrido por la crisis, los juegos no estaban a la altura de la televisión y las películas que ofrecían una historia y emociones. Películas como la “Guerras de las Galaxias” de ese entonces mezclaba efectos especiales únicos con una historia que obligaba a la persona a quedarse en su asiento para saber como terminaría (y a ver las sagas). Los videojuegos eran una actividad divertida pero no lograban definirse como entretenimiento, había que conectarse con el jugador de forma más profunda.

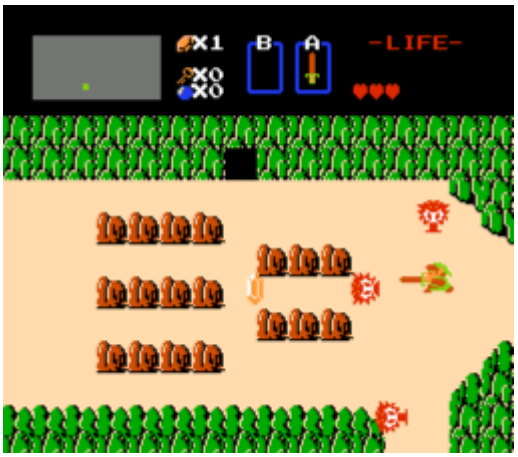
Nintendo ya existía como una compañía dentro de la industria de videojuegos ya que comercializó la Odyssey Magnavox. La evolución no la trajo un programador sino un artista japonés con habilidades para el entretenimiento llamado Shigeru Miyamoto que, inspirado en el anime y manga japoneses (dibujos que son un pilar de la cultura japonesa), creó un personaje que sería pronto más conocido que el propio Mickey Mouse, llamado Mario. Su primer juego, creado en 1981, se llamo “Donkey Kong” y Mario es el protagonista en donde enfrenta a Donkey Kong para salvar a la princesa. Debido a la popularidad del personaje Mario tuvo su propio juego en 1983 llamado “Super Mario BROS.”. Ambos juegos introdujeron el concepto de héroe, al haber un héroe los jugadores podían identificarse un poco más con ese personaje sintiéndose más atraídos a lo que le podía ocurrir, el otro concepto importante es la historia, ahora había un objetivo, un principio y un final bien definidos, si el jugador quería saber lo que iba a suceder tenía que sentarse y jugar. El “Super Mario BROS.” es el juego más vendido de la historia con 40.000.000 de unidades y es parte de la franquicia que ostenta el mismo récord con mas de 200.000.000 de videojuegos de Mario vendidos.

Donkey Kong

Super Mario BROS.

De la mano de Shigeru Miyamoto, Nintendo revolucionaría el mercado una vez más. Influenciado por lo vivido en su juventud en las afueras de la ciudad de Kyoto creó en 1987 otro héroe llamado Link, en el juego que se llama “The Legend of Zelda”. El juego tiene una historia épica, las emociones con el héroe se hicieron más fuertes, por la historia de este personaje y por la música que tiene este juego (que aún en versiones actuales se sigue escuchando). Otra característica importante es que no es un juego lineal e invita a la exploración de este mundo llamado Hyrule.

Por último y mas importante por primera vez se tiene un personaje que evoluciona, Mario es el mismo personaje tanto en el primer nivel como en el último, Link en cambio obtiene objetos, se hace más fuerte y gana habilidades. De esta forma se incluye en los juegos el concepto de Role-playing, que en ese momento existía pero sólo en algunos juegos de mesa.



The Legend of Zelda

A partir de esta época la brecha generacional que existía se fue acortando llegando al punto de que los adultos entendieron que gastar dinero (y en algunos casos mucho) solamente para jugar no era algo del todo ilógico.

2.3. La computadora entra en el mercado

Hasta ahora las consolas y los videojuegos dominaban el mercado. Los juegos de computadora no eran conocidos porque prácticamente nadie tenia una. A comienzos de los 80' Apple e IBM comenzaban la producción de computadoras de bajo costo apuntando a la familia y hogares. Roberta y Ken William fundaron en 1979 Sierra On-Line y crearon en 1983 su primer juego llamado "Mystery House". Aunque el juego carecía de sonido, animaciones y colores, entró en la historia de los juegos de computadora por tener gráficos ya que hasta ese momento los juegos estaban basados en texto únicamente.

Mystery House

En esa misma época existían computadoras personales como la TRS-80 (o más conocida como RadioShack) y la Commodore de 64 bits que luego se mejoró 128.

2.4. Salto hacía la realidad

Sega había entrado al mercado con su juego “Sonic the Hedgehog” para competir con Mario de Nintendo cuando sacó su consola Sega Genesis. Mientras Sega y Nintendo competían por el mercado con juegos infantiles con personajes y lugares imaginarios, Sony que en ese momento ya era un gigante tecnológico saca al mercado en 1994 su consola PlayStation. Pasando de los 8, 16, 32 o 64 megas que podía almacenar un cartucho de consola de Nintendo Sony alcanzo los 650 megas de almacenamiento que le proveían los CD-ROM que usaba. El salto gráfico fue instantáneo, llegaban los primero juegos en 3D. El almacenamiento de sonido y video de calidad permitió a los desarrolladores hacer juegos cinematográficos. Los juegos hasta ese momento eran épicos, futuristas o fantasiosos. Sony se dio cuenta que los jugadores que empezaron con Nintendo tenían ya entre 18 y 25 años y querían juegos más de adultos. Por eso los juegos empezaron a crearse sobre escenarios reales, seres humanos, ciudades y con objetivos realizables. Este “boom” de realidad fue tan importante que ese mismo año la Entertainment Software Association crea la Entertainment Software Rating Board (ESRB) para guiar a los padres sobre el contenido de lo que están comprando para sus hijos.

El juego más importante de esta época es el Grand Theft Auto (comúnmente conocido como GTA) es donde no hay un héroe que junta monedas y salva a una princesa sino que es un anti-héroe que sólo trata de salvarse a si mismo, comete actos criminales y es recompensado por ello. Este último punto es de suma importancia, no era tan controversial el hecho de arrollar un peatón como el hecho que se le den puntos por ello. En versiones posteriores esto fue modificado. El GTA en su quinta entrega recibió la clasificación de “AO (Adults Only)” (solo adultos) por alto contenido sexual escondido en el juego. Fue retirado de los negocios para sólo venderse en lugares específicos y sólo podía ser comprado por adultos.

GTA (1998)

GTA 3 (2001)

2.5. Actualidad

En sus comienzos los videojuegos eran consecuencia de lo que sucedía en el mundo. Un juego siempre fue “consecuencia de” ahora son “causa de”. Antes los estudios cinematográficos usaban a las empresas desarrolladoras de los mismos para publicitar más sus películas, ahora saben que el jugador que esta frente a una computadora o consola no necesariamente va a comprar una entrada de cine. Los juegos empezaron a usar actores de cine para hacer más reales e importantes sus juegos, versiones como “El Padrino”, “Star Wars” entre otros cuentan con parte del elenco original de esas películas para las voces de los personajes. En algunos casos, no son sólo voces sino el actor en carne y hueso aparece durante las cinemáticas del juego. Incluso ahora son los juegos los que son llevados a la pantalla grande como “Mortal Kombat”, “Tomb Raider” y próximamente “Halo” y “Warcraft”.

Para terminar con unos ejemplos muy interesantes, podemos tomar el caso del “Counter-Strike”, el juego FPS (First Person Shooter) más popular, que es una modificación del juego “Half-Life”. Sobre este juego se hacen torneos mundiales, Argentina compite con su propio equipo que sale de clasificaciones locales. Los premios del torneo alcanzan 100.000 dólares. En el género de del RTS (Real Time Strategy) tenemos al mejor juego de estrategia de todos los tiempo, el “Starcraft”, sobre este juego hay torneos similares a los del “Counter-Strike” y Argentina también participa. A diferencia del “Counter-Strike” que es por equipos, el “Stracraft” se juega más en duelos uno contra uno (como es el torneo). Este juego causa tanta pasión que en Corea las secundarias lo tienen como materia de taller, donde se aprende a jugar y se enseñan estrategias. El último caso es el juego más popular del momento, en el género de los MMORPG’s (Massive Multiplayer Online Role-Playing Game) con el “World of Warcraft”, posee mas de 10.000.000 de personas suscriptas alrededor del mundo que juegan diariamente pagando una cuota mensual de 15 dólares.

Counter-Strike

Starcraft

World of Warcraft

Lo más importante es que los juegos trascendieron lo suficiente como para que Universidades como la Universidad de Southern California hasta la Universidad de Central Florida, empezaron a ofrecer programas formales de diseño de videojuegos y el estudio académico de los videojuegos como parte de la cultura contemporánea. Según la Asociación Internacional de Creadores de Videojuegos, menos de una docena de universidades estadounidenses ofrecían programas asociados con videojuegos hace cinco años. Ahora, son más de 100 y hay muchas más en el exterior. En la Argentina no hay carreras en Universidades todavía pero sí las hay en Organizaciones educativas como Image Campus y la Escuela Da Vinci que ofrecen carreras en programación de videojuegos y demás áreas relacionadas como diseño gráfico y animación, otorgando incluso títulos oficiales.

2.6. Estadísticas

Las siguientes estadísticas fueron tomadas directamente de la Entertainment Software Association (ESA). A saber:

1. Las ventas sobre juegos para consolas y computadoras subieron un 6% en el 2007 aumentando a $ 9.500 millones de dólares, triplicando el número obtenido en 1996.
2. 65% de los estadounidenses juega con juegos de consola o computadora.
3. La edad promedio de un jugador es de 35 años y lleva jugando aproximadamente 13 años.
4. La edad promedio del comprador de juegos más habitual es de 40 años.
5. 45% de todos los jugadores son mujeres.
6. En el 2008, 26% de los estadounidenses que superan los 50 años de edad jugaron alguna vez un juego. Esto es 9% más desde 1999.
7. 36% de los integrantes del núcleo familiar juega en dispositivos inalámbricos con celulares y/o PDA's. Indica un 20% más que en el 2002.
8. 85% de todos los juegos vendido en el 2007 fuero clasificados con "E" (Everyone, para todos), "T" (Teen, adolescente) y "E10+" (Everyone 10+, para todos los mayores de 10 años).
9. 94% de todos los jugadores menores de 18 años indicaron que sus padres están presentes cuando compran o alquilan un juego.
10. 63% de los padres cree que los juegos son un aspecto positivo en la vida de sus hijos.

Para tomar como punto referencia y ayudar al lector a tener una idea más clara de lo que estos números representan, las ganancias reportadas por la MPAA (Motion Picture Association of America) equivalen a $9.629 millones de dólares, subieron un 5,7% con respecto al 2006 y un 10.4% con respecto a 1996.

Como se ve, las ventas anuales de ambas industrias son muy parejas y todo indica que los videojuegos pasarían a las películas en ventas en un tiempo muy cercano.

3. Marco Teórico

Se puede considerar que actualmente si pensamos en desarrollar un juego pensamos en C++, C# y Java. Decidí hacerlo en C# y a continuación se demostrará el por qué de esta elección.

Como se ve existen varios lenguajes y librerías para hacer juegos. Personalmente preferí en este caso tomar un camino más profesional y evaluar aquellos lenguajes y aquellas librerías que son mas requeridas en el mercado. Antes de dedicarme a los lenguajes más importantes quiero comenzar con una reseña sobre dos lenguajes/aplicaciones que son para hacer juegos y vale la pena mencionar.

3.1. Reseña - DIV Game Studio

Esta suite para programación de juegos incluye animaciones de sprites y edición de sonido, fue creada por la empresa española Hammer Technologies a comienzos de los 90' que permitía hacer juegos de 32-bits para MS-DOS bajo la premisa de "no es necesario saber programar". Esto no es tan así ya que el lenguaje es una mezcla de C y Pascal, ofrece muchas soluciones y facilidades pero no deja de ser un lenguaje. La estructura esta dividida en procesos que se pueden casi llamar, las variables de posición y renderización son casi automáticas, solo hay que cambiar algunas líneas de código. Estos procesos que se los pueden considerar en algún punto semejantes a las clases de un lenguaje orientado a objetos. Por ejemplo estos procesos se pueden instanciar, creando procesos únicos e independientes.

```
PROGRAM DEMO;
GLOBAL
//variable globales

LOCAL
//variables locales

BEGIN
// Se selecciona el modo de video
set_mode(m640x480);

   // Carga el archivo de gráficos necesarios en el juego
   // Los gráficos se guardaban todos juntos en lo que era como una paleta de gráficos
   // Con punteros se llama a esta colección de sprites
   load_fpg("palette.fpg");

   // Se inicializa un proceso
  proc();

END
```

```
PROCESS proc()
PRIVATE
//variable privadas
BEGIN
   x=100;        //coordenada X
   y=50;                 //coordenada Y
   z=-1;         //coordenada Z que representa profundidad
   graph=1       //el primer gráfico de la paleta

   //lógica del proceso.....

   FRAME;     // Muestra el gráfico

   END
END
```

El DIV mejoró en su versión DIV2 con la integración de características como soporte multijugador, soporte para 3D con gráficos 2D, lo que se llama Modo7, y opciones para Inteligencia Artificial. Actualmente esta aplicación no se usa por su antigüedad pero existe un proyecto nuevo llamado “Fénix” que es considerado sucesor del DIV.

En resumen, DIV es un excelente lenguaje para empezar programando juegos por lo fácil que es, igualmente se recomienda poseer conocimientos por lo menos básicos de programación.

3.2. Reseña - DarkBASIC

DarkBASIC también es una suite que ofrece IDE, compilador y debugger. Creado por una empresa británica llamada “The Game Creators”, DarkBASIC es más poderoso que DIV, trabaja sobre DirectX 8.1 para el manejo de sprites 2D y objetos 3D. Esta por supuesto basado en el lenguaje BASIC. Como se ve en el código de ejemplo a continuación y siguiendo los lineamientos del BASIC, es un lenguaje de alto nivel fácil de entender con instrucciones casi de lenguaje humano. El problema es que es un lenguaje estructura y personalmente lo considero muy difícil a la hora de diseñar un juego ya que no existe el concepto de unidad u objeto. Las variables se agrupan en arrays y el código se vuelve complicado de entender.

```
XRotate Object 1, 180
Scale Sprite 1, 100
Mirror Bitmap 1
Set Text Font "Arial"
Set Light To Object Position
If Joystick Fire A() Then GoSub PlayerShoot
```

La última versión que es de julio de este año (2008) se llama DarkBASIC Professional, posee nuevos tipos de datos y uno de ellos es un tipo de dato similar a las Structs de C.

También viene con soporte para la versión de DirectX 9.0c que es la última (sin tener en cuenta la versión 10 que trae el Windows Vista).

En resumen a DarkBASIC no le falta nada dentro de lo que son los videojuegos, se pueden conseguir juegos que compitan a nivel de los comerciales pero se necesita un nivel avanzado de programación para tratar de poner orden en un código que no provee facilidades para ello.

Cabe destacar que tanto DIV como DarkBASIC ofrecen a sus usuarios completa libertad sobre sus ganancias y códigos, alentando a los programadores a comenzar sus desarrollos y actividades comerciales sin problemas.

3.3. C++ - OpenGL - SDL

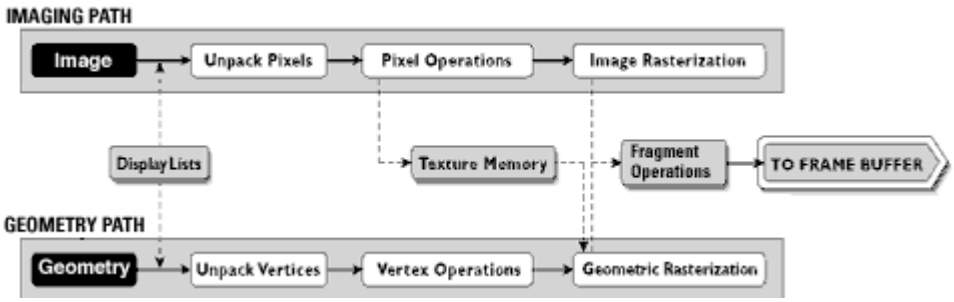
Como C y C++ son similares en rendimiento y poseen similares características pero C++ es orientado a objetos que es lo más se esta usando para programación en juegos en lo que respecta a la lógica principal del juego, para instrucciones de video se puede decidir usar C o lenguaje Assembler para mejorar la performance en esta área crítica, la evaluación se hará sólo sobre C++. Como se ha dicho, para la estructura del juego se usa C++ ya que un lenguaje estructurado cuando alcanza las 25.000 líneas de código comienza a ser difícil de mantener mientras que un lenguaje orientado a objetos sufre de ese problema recién a las 100.000 líneas de código. Por estas razones se decidió omitir el lenguaje C.

El lenguaje existe desde 1979 por eso no se puede dudar de que sea el más usado actualmente. La gran mayoría de los juegos comerciales se hacen en C++ porque la única competencia real que tiene C++ en el campo de los objetos es Java, pero como se demostrara más adelante, Java no puede incursionar en al ámbito de los juegos de computadora y mucho menos de los videojuegos.

Las grandes empresas desarrolladoras eligen C++ por su portabilidad, les permite de esta forma desarrollar un juego, y con pequeños cambios comercializarlo para PC y la mayoría de las consolas. Como se busca la portabilidad de C++, a la hora de programar juegos no se utiliza las librerías de DirectX ya que sólo funcionan en entornos Windows, por ello existe una librería también portable que es la que se usa para el manejo de gráficos llamada OpenGL (Open Graphics Library). OpenGL fue creada por Silicon Graphics en 1992 y posee mas 300 funciones para manejo de 2D y 3D que trata de simplificarle al programador la comunicación entre el lenguaje y el hardware, es la competencia directa a la librería de Microsoft Direct3D. Como DirectX es soportada por Microsoft efectúa mejoras su la librería a la par de los avances tecnológicos de las placas de video. OpenGL carecía hasta el 2006 de un grupo dedicado al soporte de la librería, la Architecture Review Borrad (ARB) le paso el control al Khronos Group (grupo del que Microsoft formó parte, dedicado a unificar API's gráficas). Esta diferencia hace que en Febrero del 2008 saliese DirectX10 con soporte para Píxel y Vertex Shader 4.0

mientras que recién en Agosto del 2008 salía OpenGL 3.0 sin soporte implícito para Píxel y Vertex Shader 4.0 pero acompañado con un plug-in desarrollado por Nvidia (compañía productora de placas de video e integrante del Khronos Group) para brindar este soporte faltante. Microsoft ya está por lanzar DirectX11 con Píxel y Vertex Shader 5.0. Esta diferencia hace que los desarrolladores (aunque por preferencias usen C++) utilicen el API DirectX, sacrificando portabilidad dedicándose solamente a hacer un juego para computadora pero que ese juego este a la vanguardia de la tecnología gráfica. OpenGL funciona de la siguiente manera:

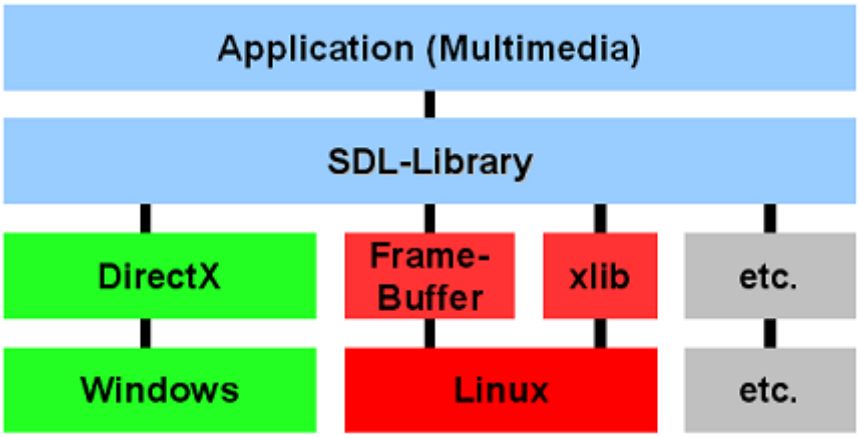
1. Del pipeline toma las imágenes (contenido 2D) y las figuras geométricas (contenido 3D).
2. Transforma imágenes en conjuntos de píxeles y las figuras en conjuntos de vértices.
3. Procesa los píxeles con funciones sobre color, coordenadas y flags para el cambio. Procesa los vectores con funciones de transformación y luces, clipping y culling entre otras.
4. Luego se rasteriza la imagen, transformando el resultado de los cambios en una imagen de bitmaps.
5. Se realizan unos últimos cambios dependiendo del los valores de profundidad que se tenían almacenados en el Z-buffer.
6. Por último los fragmentos son cargados al Frame buffer. Este buffer guarda la información del Frame que se verá luego en la pantalla.



El pipeline de OpenGL

Como se ve OpenGL es sólo un API gráfica. Por esta razón generalmente se la ve acompañada de otra librería (portable) llamada SDL (Simple DirectMedia Layer). SDL es una librería hecha en C por Sam Lantinga en 1998 con métodos que complementan a OpenGL en la parte de video agregando soporte para multi-threading y manejador de eventos, y adicionando funciones para sonido, reloj (funciones para calcular tiempo o clock), input (toma de datos desde dispositivos como teclado, Mouse y joystick), red y funciones para periféricos como impresoras y lectoras de CD/DVD-ROM. SDL se la puede ver como una capa que envuelve al sistema operativo para el manejo de las funciones antes mencionas y que puede envolver tanto a OpenGL, que es lo más común o a DirectX también, pero en este caso DirectX

es usado como backend. Este soporte con DirectX conlleva a la única diferencia que tiene con OpenGL, que es que aunque sea también portable, debe compilarse nuevamente si es cambiado de plataforma ya que posee encapsuladas diferentes funciones para cada una de ellas.



SDL como capa envolvente

En resumen, si lo que se busca es portabilidad para comercializarlo en distintas plataformas (aunque con restricciones como se verá en el apartado de C#) la unión del lenguaje C++, OpenGL y SDL es el más adecuado. La dificultad de usar estas herramientas, al no haber una organización que sea dueña y regule estas librerías, radica en la documentación que no esta centralizada y es parcial, suministrada por usuarios de las mismas y en que no se actualizan en períodos cortos de tiempo. La comunidad también aporta mejoras que muchas veces no están informadas o documentadas. OpenGL tiene lo que llama los “libros de colores” pero son de la versión 2.0. El aprendizaje sobre estas librerías debe hacerse mediante tutoriales de terceros. Lo bueno es que ambas cuentas con licencias que otorgan al desarrollador autoría y regalías completas sobre sus trabajos.

3.4. C# - XNA

C# comenzó como siendo sólo el lenguaje de la plataforma de Microsoft .NET, luego pasó a ser considerado un lenguaje estándar. Influenciado por C++, la sintaxis de este lenguaje es similar aunque en características se encuentra más cerca de Java que de C++. Ahondar en detalle sobre las diferencias técnicas escapa al objetivo de este proyecto que se focaliza en la programación de juegos. El único punto que es importante resaltar es que C# al igual que Java y a diferencia de C++ posee un Garbage Collector. La verdad de la que ya se hablaba cuando salió Java es que si uno maneja el liberado de la memoria lo puede hacer más eficiente y más rápido que el Garbage Collector. El peligro de trabajarlo manualmente es que hay que tener mucho cuidado de realmente liberar todas las asignaciones de memoria. En un sistema esto

provoca fugas de memoria, en un juego corta su ejecución, esto se debe a que una aplicación espera input de los usuarios, por lo que los ciclos son finitos mientras que un juego nunca esta "pausado" completamente, la estructura principal es un ciclo infinito, cualquier asignación o liberación de memoria mal programada, es ejecutada infinitas veces hasta obviamente el corte de la ejecución por un error.

En marzo del 2006 Microsoft publico el XNA Game Studio Framework. Un conjunto de herramientas para ayudar a desarrolladores a crear sus propios juegos para PC y para la consola de la compañía, la XBOX 360. Posee herramientas como la XACT para creación de sonidos y pistas de audio y un *Content Pipeline* propio que permite evaluar dependencias y de esta forma reducir el tamaño de un juego identificando y eliminando contenido que no es utilizado y obviamente esta de más. XNA trabaja directamente sobre DirectX, proveyendo facilidades sobre todos sus componentes. A saber DirectX esta compuesto de:

- DirectGraphics
    - DirectDraw: para dibujo 2D
    - Direct3D: para dibujo 3D
    - DXGI: para enumerar adaptadores y monitores
- DirectSound: para reproducción de archivos .wav
    - DirectSound3D: para reproducción sonidos 3D
- DirectMusic: para reproducción pistas de sonido

En su versión original 1.0 XNA no soportaba métodos para el manejo de Red, lo que restringía el desarrollo de juegos con opciones de multiplayer (multijugador) o LAN (Local Area Network). Estas funciones para red fueron agregadas en la versión 2.0 que salió a comienzos de este año (2008). Microsoft ofrece versiones gratis para sus IDE's mas conocidas (bajo el nombre de Express). De esta forma con C#, XNA y Visual C# Studio Express se puede tener un ambiente de trabajo centralizado con documentación online directamente de Microsoft y un blog llamado XNA Game Creators Club soportado directamente por Microsoft también donde MVP's de Mircrosoft ayudan a la gente inexperta con temas y tutoriales para hacer juegos.

Cambiando sólo algunas líneas de código (más que nada referentes a la parte gráfica ya que la mayor diferencia reside en donde se proyectan las imágenes) un juego puede ser portado a la consola XBOX360. Como la consola puede conectarse directamente con la computadora, un juego todavía en producción puede ser compilado y ejecutado desde la misma para ser debuggeado y testeado. Comercialmente hablando, todo juego para computadora para PC se puede vender libremente solamente respetando que en el juego no haya contenido código perteneciente a tutoriales o juegos provistos por Microsoft. Para la XBOX360 no se pueden comercializar juegos libremente sin antes haber firmado un acuerdo con Microsoft, sin embargo si se pueden publicar en el sistema online de la consola XBOX Live. Los usuarios de la consola que entran a este sistema pueden ver esta lista de juegos "alternativos" y bajárselos con intercambio de puntos. Microsoft entrega cada mes a los autores

de los juegos una suma representativa de las veces que ese juego fue bajado. Si uno de estos juegos es bajado masivamente Microsoft lo empieza a publicitar por medio de sus canales de comunicación y puede decidir venderlo como el resto de los juegos comerciales o sea, en caja, sobre estantes en locales.

La crítica más importante que recibe XNA es que usa de forma mandatoria funciones de píxel shader 1.1. Por lo tanto si creamos un juego 3D con buenos gráficos es necesario y entendible pero si queremos probar como una caja 2D rebota contra los márgenes de la pantalla estamos obligados a tener una placa de video con píxel shader 1.1. Esta crítica viene por parte de los desarrolladores que consideran esto una complicación a la hora de distribuir su juego (que muchas veces hacen juegos simples justamente para llegar a la mayor cantidad de personas posibles) y por parte de educadores que tratando de enseñar el lenguaje, el framework y/o los conceptos de programación de juegos se encuentran que quizás no todos sus estudiantes poseen los ingresos como para estar comprando placas de video nuevas. Este problema también es una desventaja importante en computadoras portátiles ya que la mayoría carecen de una placa de video comercial pero poseen el chipset GPU (graphics processor unit) de alguna de estas marcas, pero con la diferencia es que el soporte de píxel shader es emulado y XNA no lo detecta.

En resumen, este conjunto del lenguaje C# y el Framework XNA forman un ambiente de trabajo propicio para aquellos que quieren empezar por su facilidad y para aquellos que ya son avanzados y piensan en esta tarea que hasta ahora era un hobby en una actividad comercial rentable. Todavía no hay una gama de juegos comerciales importantes desarrollados con estas herramientas pero se puede entender por la poca vida que lleva pero debido al soporte brindado por Microsoft para PC y para su consola, se espera que esto empiece a cambiar.

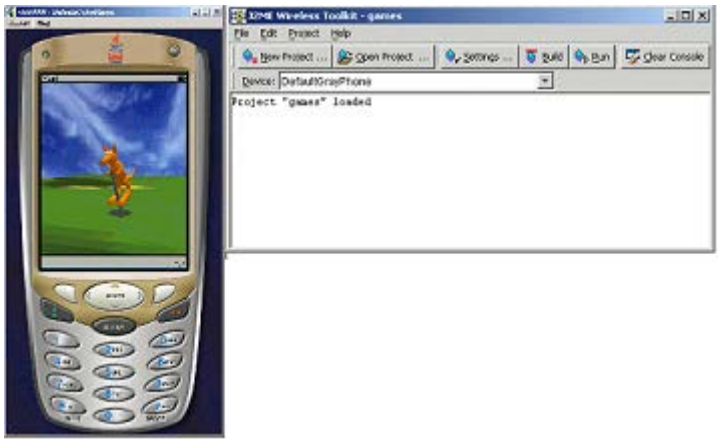
3.5. Java - J2ME - MIDP

La portabilidad de Java es indiscutible gracias a su estructura de instrucciones en forma de bytecode que son interpretadas por cualquier plataforma que posea la Java Virtual Machine. Igualmente como sólo nos interesa en este proyecto la relación de Java con el desarrollo de videojuegos debemos decir que Java no es usado para los mismos, el único juego conocido (no comercial) hecho en Java para PC se llama “Runescape” que es un MMORPG, su popularidad no es dada por el aspecto gráfico del mismo o por su jugabilidad sino mas bien, porque funciona en cualquier explorador de Internet. Esto permite a las personas poder jugar virtualmente desde cualquier máquina. Hay 15 millones de usuarios registrados en este juego que tiene una versión gratis (con publicidad incorporada) y una versión paga en donde los usuarios disponen de más contenido.

Runescape (2da version)

La relación más importante que tiene Java con la industria de los videojuegos es el desarrollo de los mismos para celulares gracias a su especificación J2ME o también llamada Java 2 Micro Edition. Esta configuración más reducida que su padre J2EE posee es una colección de API's para interactuar con el hardware de dispositivos móviles. Siendo Java el lenguaje en su forma de J2ME, la librería que seria el equivalente al SDL o a OpenGL se llama MIDP (Mobile Information Device Profiles, sólo para celulares). Se encuentra en su versión 2.0 y provee funciones que la describen como un API's específica para juegos. La complicación en el desarrollo para estos juegos no es en el juego en sí pero a diferencia de una consola, en la que uno ya sabe los componentes de hardware que posee y el entorno donde se encuentra o en el caso de la computadora en donde pueden existir si se puede decir infinitas configuraciones pero hay estándares y se trabaja generalmente sobre entorno Windows y conociendo los drivers de las placas de video pero sabiendo que arriba se encuentra la capa de DirectX que es igual en todas la ellas, en el caso de los dispositivos móviles las configuraciones son realmente infinitas y cada fabricante opta por usar ciertas piezas de hardware o distintos lenguajes. Un programa hecho en MIDP 2.0 acota sus posibilidades de venta sacando a posibles compradores que poseen celulares con Java pero MIDP 1.0, y a dispositivos con BREW (Binary Runtime Environment for Wireless de Qualcom) o Symbian OS (de Nokia) que soportan Java pero para obtener sus API's hay que pagar licencias. Es así que para un juego destinado a los dispositivos móviles se encuentran muchas versiones del mismo, incompatibles entre sí que deben ser mantenidas si se desea abarcar la mayor parte mercado posible. Se espera que una empresa que se dedique a estos desarrollos ya posean Frameworks para que sus juegos sean fácilmente portados de un dispositivo a otro.

El desarrollo para estos juegos puede ser tan poco como un mes y nunca superan los seis meses, se consideran proyectos chicos, y gracias a la documentación provista por Sun y de tutoriales es fácil hacer juegos para celulares. Otra gran ventaja es que existen emuladores (el más importante es el J2ME Wireless Toolkit provisto por Sun) para diversos celulares o PDA's, esto ayuda al programador a ir testeando el juego durante la etapa de desarrollo que es algo que para los juegos de consolas es más difícil de lograr o al menos más tedioso.



J2ME Wireless Toolkit ejecutando un juego

En resumen, Java no posee el suficiente rendimiento para correr un juego de PC estándar, esta totalmente afuera del mercado de las consolas ya que las mismas no poseen una Java Virtual Machine. Tiene su mercado en la industria en lo que son los celulares principalmente, J2ME es fácil de usar y sirve para aprender los conceptos básicos en lo que programación de juegos, lo aprendido con el lenguaje por su orientación a objetos hacen que esos conocimientos se puedan aplicar a la hora de hacer un juego más complejo en C++ o C#. La comercialización de estos juegos se puede lograr pero con ganancias mínimas ya que al principio dejaremos a la mayoría de los consumidores afuera y cada juego se cotiza entre $1 y $5 pesos como máximo. Como este proyecto se trata sobre un juego de más envergadura, Java y J2ME quedaron descartados ya que en Java no se puede lograr como se pretende un juego complejo y un juego para celulares lo considero un proyecto chico.

4. Métricas

En este apartado vamos a dejar lo teórico a un lado y enfocarnos más en lo práctico y en los resultados obtenidos a partir de una prueba. Para obtener los siguientes datos se hizo un mismo programa que calcula los FPS (frames per seconds) de la aplicación en los tres lenguajes que se vienen analizando para saber de modo tangible como se comporta cada uno de ellos y obtener algunas mediciones extras.

4.1. El programa prototipo

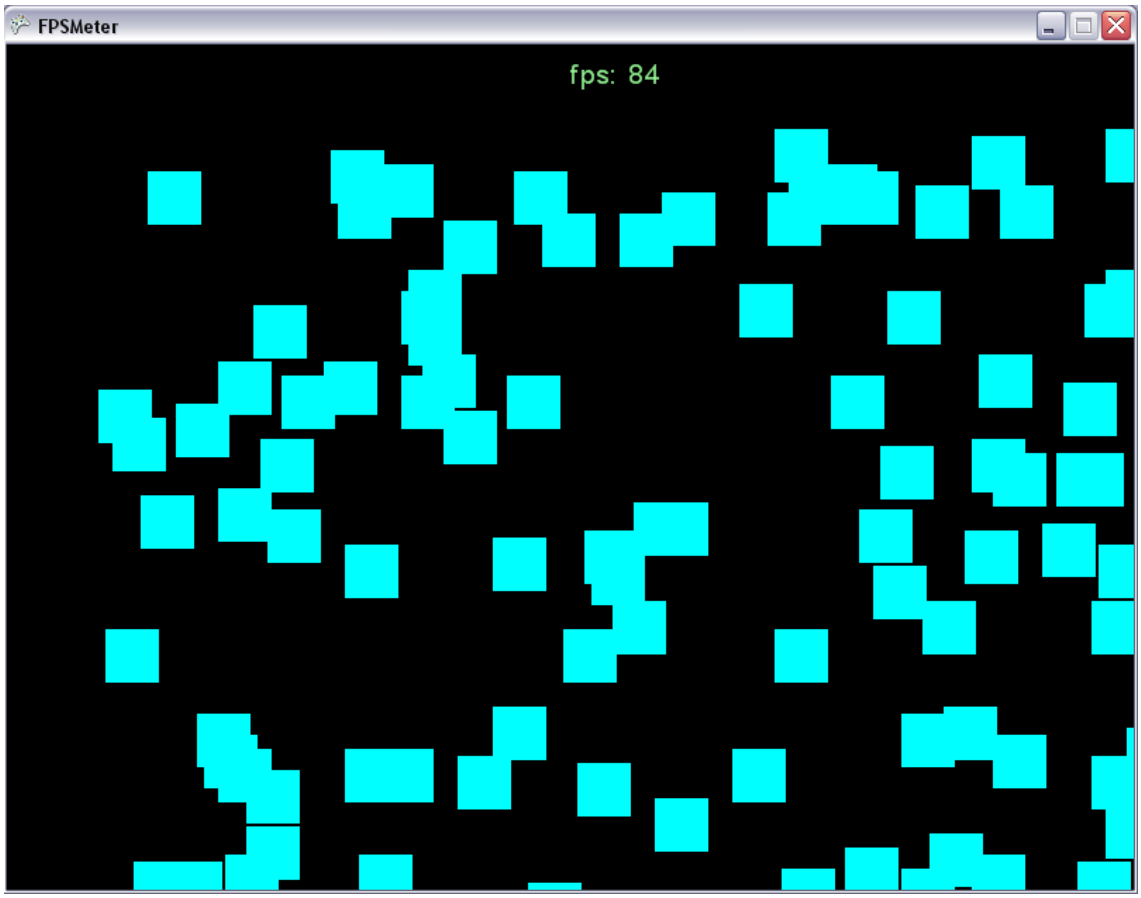
La aplicación en reglas generales consiste en:

1. Una fase de iniciación donde se carga un archivo de imagen .bmp (bitmap) llamado rec.bmp.

rec.bmp

2. En esta misma fase se crean tantos objetos “Rectángulo” como los indicados por otra variable que indica la cantidad esperada de los mismos. Cada uno de los mismos se crea en una posición al azar dentro de la ventana fijada en 800x600 píxeles. Todos se mueven a la misma velocidad y empiezan con una dirección hacia abajo y a la derecha.
3. Una vez dentro del Loop principal del juego estos rectángulos rebotan contra los límites de la ventana donde corre la aplicación de forma automática.
4. Dentro de este Loop se encuentra la función *calculateFPS()* que mientras se este en un mismo segundo de ejecución va sumando la cantidad de Frames (o ciclos) que pasaron. Cuando el segundo cambia, se muestra la cantidad de Frames que hubo y el conteo vuelve a cero, para empezar a sumar para el segundo siguiente. A continuación el código de esta función para su versión de C++ con SDL:

```
void CApp::CalculateFPS()
{
    newTime = SDL_GetTicks();
    if (newTime - oldTime < 1000)
    {
            fps = fps + 1;
    }
    else
    {
        cout << fps << '\n';
        oldTime = SDL_GetTicks();
        fps = 1;
    }
}
```



FPSMeter (versión C# - XNA, con 100 rectángulos)

4.2. Parámetros

Se evaluaron los siguientes parámetros:

1. **Líneas de código:** a fin de que la medición sea lo más parcial, precisa y justa posible, los tres códigos fueron formateados exactamente de la misma forma, incluso no respetando las *"Best Practices" (mejores prácticas)* de cada uno de ellos.
   Por ejemplo: al escribir una función en C# la llave que abre la definición del código se ubica abajo del nombre de la función mientras que en Java se coloca a la derecha del mismo. Se ubicó en Java la llave inicial debajo de la función aunque no es lo que se hace cuando se programa en este lenguaje, pero sino estaríamos teniendo una diferencia de una línea en cada función, clase, condición o ciclo.

2. **Cantidad de archivos:** se respeto la estructura que recomienda cada lenguaje o mejor dicho cada librería para la creación de un juego. No se crearon clases extras más allá de lo que se considera una *"Best Practice"* para cada lenguaje. Por ejemplo: XNA tiene una clase inicial llamada Program.cs que se crea automáticamente, SDL crea un

archivo por cada etapa dentro del ciclo de un juego y tenemos los archivos de librería .h; y Java es la única que al no estar apuntada hacia juegos no tiene una estructura definida.

3. **Horas/hombre:** se asentaron las horas dedicadas a la creación de cada prototipo. Se tomó en cuenta para la medición que el conocimiento de cada uno de ellos no era igual ya que después de la creación del juego, C# es el lenguaje que en este momento mejor manejo, Java fue visto en la facultad y he trabajado en varias oportunidades con él y en cambio C++ era la primera vez que lo usaba. Las horas/hombre incluyes el período de aprendizaje de la librería y el tiempo que se tardo efectivamente en hacer la aplicación.

4. **Tamaño del proyecto:** este parámetro toma en cuenta el tamaño total en disco que ocupa el proyecto, es decir, su código ejecutable más todos los archivos que cada lenguaje y/o librería agregue. Ya que la idea es que nuestro juego lo jueguen otros es importante saber el tamaño de lo que queremos distribuir.

5. **Memoria en uso:** se tomó nota sobre la cantidad de memoria RAM (en KB Kilobytes) que cada aplicación utilizó durante su ejecución así también como las variaciones que presento durante la misma.

6. **Uso de CPU:** se tomó nota también del uso de CPU (en porcentaje como lo muestra Windows) para saber como afecta cada lenguaje al procesador.

7. **FPS (Frames Per Second):** el parámetro más importante ya que estamos hablando de juegos. La velocidad de ejecución tiene que ser la prioridad y en definitiva es lo único que le interesa al jugador. El valor ideal de este parámetro es 60fps. Este número asegura una transición suave entre frame y frame que es lo que se considera comúnmente como *"el juego corre bien".* Como información adicional, el ojo humano es incapaz de ver más de de 72fps así que se considera ese el tope máximo y como mínimo se esperan 30fps que es la velocidad de reproducción de una película en el cine.

4.3. Resultados

Para la medición de **Memoria en Uso**, **Uso de CPU** y **FPS**, se corrieron tres pruebas cambiando la cantidad de los cuadrados mostrados en pantalla. Esto se utiliza para ver cuanto se le puede exigir a cada lenguaje. Por eso en la plantilla de resultados que se puede ver a continuación, cada uno de estos campos esta marcado con la cantidad de rectángulos que se usaron; 100, 1000 y 10000 respectivamente para cada prueba.

Estas pruebas se corrieron en una PC con Windows XP, Intel Core2 Duo 2,4GHz, con 2GB de memoria RAM y placa de video NVIDIA GeForce 7950GT de 512MB de memoria.

	C++ - SDL	**C# - XNA**	**Java**
Líneas de Código	243	182	198
Cantidad de Archivos	CApp.h CApp.cpp CApp_OnCleanup.cpp CApp_OnEvent.cpp CApp_OnInit.cpp CApp_OnLoop.cpp CApp_OnRender.cpp Rec.h Rec.cpp TOTAL: 9	Program.cs Game1.cs Rec.cs TOTAL: 3	FPSMeter.java Vector2.java Rec.java TOTAL: 3
Horas/hombre	18hs.	2hs.	8hs.
Tamaño del proyecto	FPSMeter.exe - 846KB rec.bmp - 5KB TOTAL: 851KB	FPSMeter.exe - 24KB rec.xnb - 6KB TOTAL: 30KB	FPSMeter.jar - 11KB rec.bmp - 5KB TOTAL: 16KB
Memoria en uso (100)	5,412 KB	15,568 KB	17,832 KB
Uso de CPU (100)	48 - 50%	2 - 7%	2 - 6%
FPS (100)	718 - 729	84 - 85	82 - 84 (0)
Memoria en uso (1000)	5,504 KB	15,648 KB	18,936 KB
Uso de CPU (1000)	49 - 51%	39 - 44%	21 - 27%
FPS (1000)	141 - 149	74 - 78	82 - 84 (1 - 2)
Memoria en uso (10000)	5,712 KB	16,156 KB	19,628 KB
Uso de CPU (10000)	49 - 51%	44 - 48%	63 - 69%
FPS (10000)	17 - 18	8 - 9	17 - 18 (4 - 7)

Cuadro comparativo

4.4. Comentarios Post-prueba

4.4.1. C++ - SDL

- **Líneas de código:** SDL propone una estructura acorde para la programación de juegos, que abarca lo que es el Loop principal del mismo, con la diferencia que cada etapa esta ubicada en un archivo o clase separada. De esta forma la clase principal del programa incluye una librería .h que contiene todas estas funciones que van a estar definas en archivos separados que obviamente incluyen esta librería.

Entonces encontramos para la inicialización de variables el método/archivo *OnInit*. Para lo que se considera como la lógica del juego que se resuelve de forma automática, como el movimiento continuo de un objeto o procesar Inteligencia Artificial tenemos el método *OnLoop*, para la actualización de la lógica del juego pero por parte del usuario, lo que se considera como input existe el método *OnEvent* que toma justamente input del teclado, Mouse o joystick y/o cualquier otra cosa definida como evento, y aquí es donde podemos definir que queremos que se haga cuando uno de estos eventos sucede. Como parte del Loop tenemos el método *OnRender* que contiene la lógica para la renderización del juego en pantalla. Ya fuera del Loop tenemos un último método llamado *OnCleanup* que contiene el código para la liberación de memoria antes de cerrarse el programa, en C++ aparte de los *constructores* debemos usar los *de-constructores* para este fin. Debido a la cantidad de archivos, a la separación del mismo y a que a diferencia de C# o Java tenemos que administrar nuestra memoria manualmente no era raro esperar que el código en C++ sea aproximadamente un 25% mas extenso que en otros lenguajes.

- **Cantidad de archivos:** entre la separación de funciones que hace SLD para el Loop principal más que a cada clase le corresponde un .h que la defina tampoco era raro pensar que el número de archivos pueda duplicar al de los otros lenguajes.

- **Horas/hombre:** el aprendizaje de C++, de SDL, del hecho que C++ no sea totalmente orientado a objetos sino que se lo considere un híbrido, la posibilidad de escribir tanto a alto como a bajo nivel, depender de librería externas para facilitar la programación (SDL para juegos en este caso), mensajes de error poco específicos y la falta de un IDE tan potente como Visual C# o NetBeans hacen que la programación en este lenguaje sea poco automatizada y obliga a tener un amplio conocimiento de las clases y sus métodos para tener una programación sin pausas. Es sin dudas el lenguaje más complicado de programar. La cantidad de archivos, la falta de un Garbage Collector y la necesidad de programarlo uno mismo, hace necesario más líneas de código y por ende más tiempo que en otros lenguajes para programar lo mismo (el medidor de FPS en este caso).

- **Tamaño del proyecto:** el ejecutable resultante es notablemente más grande que el equivalente a los otros lenguajes, esto se debe a que como lenguaje compilado, contiene todas las librerías que se usaron. Los elementos multimedia no poseen ningún tratamiento especial y ocupan en disco exactamente el tamaño que poseen.

- **Memoria en uso:** la eficiencia con la que C++ maneja la memoria se hizo notar en este punto, además se observó durante las pruebas que la memoria no variaba (o lo hacia

mínimamente) recordándonos que el manejo de la memoria no es automático y lo manejamos nosotros. Se puede decir que ni C++ ni SDL usan un byte de más de los que están programados. Tener tal control en un juego puede ser tedioso para programar pero se le ve su lado ampliamente positivo ya que sabemos exactamente como se comporta y cuantos recursos utiliza.

- **Uso de CPU:** a lo largo de las 3 pruebas el uso de CPU fue constante, alrededor del 50%. Este uso puede ser excesivo en el caso de la primer prueba en donde C# y Java decidieron sólo usar lo indispensable para correr las pruebas, solo aproximadamente un 4% de CPU. Como estamos hablando de juegos es muy extraño encontrar uno corriendo al mismo tiempo que en background haya una aplicación igual de exigente en cuanto a los recursos. Jugar dos juegos al mismo tiempo no es práctico, o por ejemplo tener un Oracle haciendo un backup en frío mientras estamos jugando es simplemente ilógico por lo que este valor no debe considerarse negativo y además un uso del 50% deja exactamente la otra mitad libre para que Windows y aplicaciones de menor envergadura operen libremente.

- **FPS:** C++ superó a C# y a Java en una relación casi 2:1 (se explicará mejor esta relación en las conclusiones de C#) en los FPS que logró. Su programación de nivel medio, no incorporación de funciones automatizadas y que sea un lenguaje compilado logran que realmente C++ se destaque velocidad de procesamiento.

4.4.2. C# - XNA

- **Líneas de código:** las facilidades que ofrece XNA nos dan como resultado el código más corto de los tres lenguajes. El archivo autogenerado, Game1.cs (al que le podemos cambiar el nombre) posee una estructura con métodos ya definidos que conforman el Loop principal del juego, a saber: comienza con el constructor de la clase, el método *Game1()* donde se define el dispositivo de salida (la pantalla) y su configuración, continua con el método *Initialize()* para definición de variables, el método *LoadGraphicsContent()* para lo que es carga de contenido multimedia (imágenes, fuentes, sonidos, videos, etc). Este último método es particularmente importante ya que la memoria en una placa de video no esta tan protegida como la memoria de la computadora y cada petición de escritura en el buffer de la GPU (Graphics Processor Unit) puede pisar algo ya existente, por ejemplo: la imágenes de un juego están cargadas en el buffer pero se ejecuta otra operación de escritura como la del salvapantallas, se corre peligro de que parte de la información se pierda. Llegado a ese caso extremo XNA vuelve a ejecutar el método *LoadGraphicsContent()* para insertar en el buffer gráfico toda la información y restaurarla. Le sigue un método denominado

UnloadGraphicsContent() que sin necesidad de redefinirlo se encarga de vaciar de la memoria todo lo cargado en el método *LoadGraphicsContent().* Es una facilidad que nos ofrece XNA de dejar que este proceso este automatizado, igualmente si creemos que podemos mejorar esto o tener más control sobre el mismo podemos agregar nuestro código al método. Ya parte del Loop principal tenemos el método *Update()* que se encarga de la lógica principal de juego, tanto la automática como el manejo del input por parte del usuario, es común hacer que este método llame a otro definido por nosotros que puede ser llamado *UpdateInput()* y así separar ambas lógicas. El último método es el *Draw()* que se encarga de mandar al dispositivo de salida los elementos que definamos en esta sección. El Loop en XNA no esta definido como tal, es invisible al usuario, *Update()* y *Draw()* se llaman en ciclo pero no hay ninguna sentencia al estilo *"while(true)"*. *Initialize(), LoadGraphicsContent(), Update() y Draw()* en un ciclo, y finalmente *UnloadGraphicsContent()* no son explícitamente llamadas, esto es invisible al usuario y le da una facilidad más a la hora de programar.

- **Cantidad de archivos:** por la estructura del XNA siempre vamos a tener mínimos dos archivos que son creados automáticamente que son el *Program.cs* y el *Game1.cs* (nombre por default). A partir de esta base por cada clase se creara otro archivo adicional .cs.

- **Horas/hombre:** el aprendizaje de XNA obviamente tomó su tiempo pero con una gran cantidad de tutoriales, incluso videos se torno fácil, lo que tomó algo mas de tiempo fue acostumbrarse a C# en sí aunque su similitud con Java también fue una ventaja. Con un lenguaje completamente orientado a objetos, un IDE que posee ayuda en sí mismo y opciones de autocompletado, la aplicación hecha en FPSMeter fue sin duda la más fácil y rápida de hacer, incluso cuando fue la primera y hubo que pensar la estructura y la función que mide los FPS.

- **Tamaño del proyecto:** el ejecutable resultante es de poco tamaño ya que a pesar de que XNA es una librería extensa se pueden definir específicamente que librerías se pueden usar. Mientras que en C++ se define la librería SDL.h, en C# se puede incluir XNA.Framework.Graphics solamente, por ejemplo. El contenido multimedia es transformado a otro tipo de archivo .xnb que generalmente funciona como método de comprensión también (no en el caso de imágenes) pero que su función principal es proteger nuestros archivos para que no sean usados/modificados por otras personas.

- **Memoria en uso:** siendo el triple de C++ y apenas inferior que Java se puede entender por todos lo métodos automatizados que ejecuta el XNA. La memoria sufre un incremento recursivo a lo largo de la ejecución.

- **Uso de CPU:** el uso de la CPU es un balance entre los FPS mínimos esperados y el nivel de procesamiento requerido para ello. Como máximo siempre se mantiene alrededor del 50%.

- **FPS:** siguiendo el hilo sobre el uso de CPU se puede entender por que tiene tan bajos FPS en comparación con C++ en la prueba de 100 objetos, no encuentra necesario usar más del 4% de CPU para tener un framerate más que aceptable. Por lo observado en las pruebas de 1000 y 10000 donde C++ duplica en FPS, se puede suponer que si no tuviese una restricción de Uso de CPU (o FPS máximos en realidad) la prueba con 100 objetos estaría devolviendo unos 350 FPS. El límite implícito en los FPS no es algo malo, todo lo contrario, un juego que posee FPS excesivamente tan altos es no se puede jugar y esto requeriría que programemos un algoritmo para restringir la velocidad de procesamiento.

4.4.3. Java

- **Líneas de código:** al no tener facilidades y/o guías para la programación de juegos, el código no tiene ninguna estructura especial aunque se respetaron las convenciones básicas del Loop principal. Varias cosas a tener en cuenta sobre el código de Java, aunque no es el de mayor tamaño, es el que requirió hacer más cosas manualmente. Los juegos por necesidad corren en un thread (hilo de ejecución) para que todas las acciones de update, input y renderización se asemejen más a una ejecución paralela que a una estructurada. Mientras que XNA y SDL esconden esta característica, en Java hay que programarla uno mismo por ende la clase principal de nuestro programa implementará la interfaz *Runnable* que maneja los threads. Nuestro dispositivo de salida se define en la clase principal misma que va a heredar de la clase Frame (puede heredar o poseer un objeto Frame, es indistinto). Como nuestra clase es un Thread y un Frame tenemos dos métodos claves, el método *Run()* para iniciar el thread y el método *Paint()* para dibujar lo que posea el Frame. Como parte de las características de Java el método *Paint()* no debe llamarse directamente sino que por medio de otros dos métodos que son el *Repaint()* y el *Update()*, la lógica de estos métodos conlleva varios problemas que serán vistos a la hora de calcular los FPS. Otro punto que no esta contemplado en Java pero si en C# y C++ y que hubo que programar para que la evaluación del rendimiento sea pareja fue la del DoubleBuffering. Esta técnica elimina lo que se conoce como *Flickering.* Como un juego va sobre un Thread la renderizacion en pantalla no se da en el mismo momento, o sea en un momento de tiempo no se consigue un Frame donde todos los objetos estén ya en su nueva posición. Por más que estemos hablando sobre milésimas de segundo el ojo humano es capaz de notar

como algunos objetos se renderizan antes que otros dando una sensación de que aparecen y desaparecen, a esto se le llama *Flickering (parapadeo)*. El DoubleBuffering propone no renderizar los objetos en el dispositivo principal de salida sino en uno auxiliar ubicado fuera de pantalla, cuando todos los objetos fueron dibujados se lleva todo ese Frame en Offset a la pantalla principal dando la sensación buscada de que todos los objetos son dibujados al mismo tiempo por Frame.

- **Cantidad de archivos:** como se explicó anteriormente la falta de una estructura para videojuegos hace que la cantidad de archivos sea menor. Se reportaron tres archivos pero se puede decir que el la clase Vector2.class fue creada solo para facilitar el manejo de las coordenadas y no estrictamente necesaria, pudiendo llevar el número de cantidad de archivos a solo dos.

- **Horas/hombre:** como proyecto fue relativamente fácil de lograr, conocimientos previos, lenguaje orientado a objetos y un IDE completo con NetBeans hicieron que la programación sea más ininterrumpida que C++. A lenguajes de similar dificultad entre C# y Java la diferencia de tiempo se debió a que no se encuentran tutoriales para el desarrollo de videojuegos y los distintos aspectos que se deben programar en Java y en C# no como los Frames, Threads y hasta DoubleBuffering hay que tratarlos individualmente y dedicarles tiempo a cada uno de ellos.

- **Tamaño del proyecto:** el archivo .bmp de imagen es similar en todos, en Java no sufre ningún tratamiento especial. Ahora lo que es el programa en sí es considerablemente inferior que el de los otros lenguajes, primero por la agrupación del proyecto dentro de un archivo .jar y luego porque en esta compilación no están las librerías ya que son propias de Java y se pueden resolver con la JVM (Java Virtual Machine).

- **Memoria en uso:** el consumo de Java es el mayor de los tres lenguajes superando en unos megabytes a C#, pero hay que aclarar que aparte de la aplicación en sí, ya están siendo ocupados entre 20 y 40 megabytes de memoria para la JVM que no fueron reportados en los resultados.

- **Uso de CPU:** Java utiliza del procesamiento de la CPU solamente lo necesario aunque sin restricción alguna, si esto fuese una aplicación mucho más exigente y no un prototipo para unas pruebas llevaría el uso de la CPU a valores cercanos al 100% que perjudicaría el funcionamiento en background del resto de los procesos.

- **FPS:** este es el punto más problemático de este lenguaje. A diferencia de los otros datos los FPS en Java no son absolutos sino que por el contrario son muy relativos. Como se sabe el llamado al método *Paint()* es automático y incluso a través de los métodos *Repaint()* y *Update()* las llamadas no siempre son obedecidas ya que estas se ejecutan también de forma automatizada. *Paint()* se ejecuta automáticamente cuando el ciclo de ejecución *considera* que hubo un cambio notable en el Frame y es necesario renderizarlo devuelta. De esta forma Java considera no necesario dibujar el Frame cada ciclo. Al usuario le queda forzar este llamado usando el *Repaint()* pero esto lo único que logra es ubicar un pedido en la cola de ejecución del Thread para ejecutar el *Update()*. Este pedido puede no concretarse nunca ya que Java correrá el Thread sólo si tiene el tiempo "de sobra" como para poder ejecutarlo. Es por eso que dentro de la programación del Thread para nuestro juego generalmente ponemos al final una instrucción llamada *Thread.Sleep*() que toma como parámetro un tiempo en milisegundos. Al obligarle a Java a detener la ejecución del Thread principal lo inducimos a que use esa pausa para correr el Thread pendiente del *Update()* que termina por llamar al *Paint()* que es lo que queremos. En el programa hecho en Java para esta prueba se llegó a la conclusión que 12 milisegundos es un tiempo adecuado para lograr este balance entre el procesamiento de la lógica y del dibujo. Igualmente y como se ve ese balance es muy difícil de conseguir. Los resultados de los FPS para Java poseen un número en entre paréntesis, esa es la cantidad de ciclos que Java en promedio ejecuta sin dibujar, entre cada *Paint()* efectivo. Aumentando los milisegundos para el *Sleep()* reduce esta brecha pero baja los FPS considerablemente. De todas formas no es posible saber cual es el promedio adecuado porque si subimos demasiado el tiempo de espera estaríamos bajando los FPS y la ejecución en general sin razón. El poco control que Java ofrece sobre una parte tan importante como la renderización hace que flaquee el interés de programar videojuegos en este lenguaje. Como los ciclos aunque no se dibujan si se procesan, un objeto termina en el mismo lugar como si estuviese hecho en otro lenguaje, e incluso más rápido pero en vez de haber logrado una sensación de movimiento el objeto va "haciendo saltos" por la pantalla.

4.5. Conclusiones

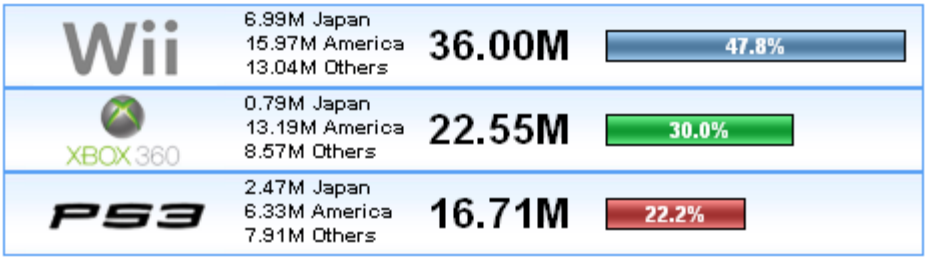
Empecemos con Java. No fue particularmente difícil hacer el programa, existe mucha documentación y soporte por parte de Sun Mircrosystems. Aunque requirió programación adicional para funciones que los otros dos lenguajes ya implementaban automáticamente como DoubleBuffering o la lógica detrás del hilo principal de ejecución no derivo en mayores complicaciones. En cuanto al resultado obtenido, se notó que Java es un lenguaje de rápido procesamiento, aunque a costa de no limitarse en el uso CPU, esto se puede observar en

programas complejos como el Azureus (cliente bittorrent), que luego de un tiempo prolongado de ejecución llevan el Uso de la CPU al 100%, casi inutilizando a la computadora. Su parte gráfica se considera extremadamente eficiente para programas administrativos en donde la llamadas al *Paint()* están ligadas a los conocidos *ActionListeners* y sólo cuando hay un clic o algún otro tipo de input que podría generar un cambio en la pantalla se ejecuta. En cambio, para este caso que nuestro foco esta en los videojuegos, no podemos tolerar no tener el control en un proceso tan importante como la renderización, que los objetos actualicen sus variables pero que estos cambios se reflejen sólo en algunos Frames hace imposible tener un juego con un funcionamiento coherente. Además aún obteniendo renderización en cada Frame y con el DoubleBuffering activado, la parte gráfica no iguala en calidad la que podemos llegar a encontrar con C++ o C#, el movimiento es más suave en estos últimos. Si hay que asociar a Java con videojuegos, lo haremos con J2ME y MIDP para celulares. La comparación no se hizo con esta configuración porque no era equivalente comparar un juego para PC o consola con uno para celular. Por estas razones Java no fue elegido para este proyecto.

Para el caso de C++, debemos decir que es cierto que C++ es el lenguaje buscado en el mercado para desarrollo de videojuegos. Su característica de nivel medio posibilita el desarrollo de la lógica general del juego en C++ y permite el desarrollo de módulos de alta importancia, como los que se pueden encontrar dentro del *Engine* gráfico para mejorar su rendimiento, en código Assembler. Posee una velocidad de aproximadamente del doble de C#, aunque esta es un ventaja requiere de programación adicional para restringir a 60 u 80 los FPS ya que a valores más altos el juego sería imposible de jugar. Su sintaxis mitad del lenguaje estructurado y mitad lenguaje orientado a objetos dificulta su programación. El manejo de la memoria es preciso y esto es un atributo valorado en los juegos pero funciona como arma de doble filo si esa precisión no se encuentra también al mismo nivel en cuanto a la liberación de la misma. Una sentencia de liberación de memoria no programada puede ocasionar un overflow de memoria, aunque hablemos de una porción mínima recordemos que estamos dentro de un ciclo “infinito” de ejecución, porciones de memoria son utilizadas pero no liberadas en número creciente hasta que alguna de estas peticiones trata de acceder a un segmento protegido por falta de memoria libre. La liberación de memoria post-ejecución (o sea fuera del ciclo) no es critica para la aplicación pero queda a conciencia del profesional liberar los recursos que ya no va a ser utilizados para facilitarle al sistema operativo y al usuario el uso de la computadora. SDL y C++ no son una mala combinación pero por tener una facilidad de uso menor y porque C# tiene algunas prestaciones más que se verán a continuación, C++ no fue elegido como lenguaje para este proyecto.

C# y aún más XNA tienen pocos años en el mercado, se reconoce en forma general que un lenguaje orientado a objetos es más adecuado para la programación de videojuegos que uno estructurado por la facilidad de programación, visualización y mantenimiento del mismo. C# posee una velocidad de procesamiento adecuada ya que se vio que es capaz de procesar 1000 objetos a una velocidad mayor a 60 FPS que se considera lo estándar. Un juego

con más de 1000 objetos en escena debe considerarse algo como un juego comercial de primera línea. A iguales rendimientos, lo que terminó por convencer de este lenguaje es una posibilidad que los otros lenguajes no ofrecen, la posibilidad de “comercializar” nuestro juego a través de una consola. XNA, C# y la consola XBOX360 son de Microsoft, los videojuegos hechos con XNA puede ser (con algunas modificaciones) ejecutados en la XBOX360. Para ver la magnitud de esto, a continuación se presentan las ventas reportadas hasta el momento de cada consola.



Ventas de consolas por unidad (en millones)

Como se ve la XBOX360 ocupa la segunda posición. Para programar para la Wii y la PS3 es necesario ser Partner Desarrolador de Nintendo y Sony respectivamente y comprarles a ellos los frameworks para programar para esas consolas. Todo el desarrollo del proyecto se llevó a cabo sin realizar ningún tipo de pago, todos los elementos utilizados son gratuitos y libres de utilizar, por lo que pagar para desarrollar para una consola en este punto no es compatible con la filosofía que venimos llevando. Para XBOX360 no sólo que no tenemos que pagar sino que aprovechamos la distribución gratuita de nuestros juegos por el sistema XBOX Live de la consola. Cada cierto tiempo Microsoft eleva un reporte con los juegos más descargados de este servicio ofreciéndole al creador del mismo una remuneración monetaria, aquellos juegos que se destaquen por estar siendo jugados masivamente por otros usuarios pueden por parte de Microsoft ser vendidos como los videojuegos comerciales o sea en locales afines, en donde la remuneración para el creador será un porcentaje de la ventas, beneficiándolo mucho más al mismo (y obviamente a Microsoft). XNA y C#, por su facilidad y porque es la única combinación que ofrece un *port* a una consola y un inicio en el mundo comercial fue elegida para el desarrollo del proyecto TEGE.

5. Conceptos sobre los videojuegos

En esta sección veremos la estructura principal, los componentes y demás elementos, conceptuales y de programación, necesarios para programar un videojuego. Por un tema de los límites del proyecto, se va a omitir la teoría detrás de los elemento de IA (Inteligencia Artficial), lo denominado multi-jugador o redes y todo lo relacionado con física y partículas.

5.1. El Loop principal

A diferencia de un script o programa que contiene una secuencia de instrucciones que se ejecutan de forma casi lineal o en el caso de un sistema en donde la ejecución es llevada por las acciones de un usuario, un juego ejecuta sus sentencias una y otra vez ya que su código principal o *Core Engine* se encuentran en un loop que se corta solamente cuando salimos del juego. En pseudo-código la espina dorsal del mismo es de la siguiente forma:

```
Game()
{
    Initialize();
    Load();
    While (!exit)
    {
      Update();
      UpdateInput();
      Render();
    }
    Unload();
}
```

1. **Initialize():** al estar fuera del Loop esta sentencia se ejecutan una sola vez. En esta sección inicializaremos las variables definidas en la clase principal, en el caso de variables simples se pueden definir ni bien se crean, en el caso de objetos quizás la lógica del programa nos invita a ser más ordenados. Crear un objeto significa, instanciar una clase. Como poseen un constructor muchos atributos son generalmente pasados al constructor haciendo parecer esta sección del juego poco útil pero no es así, *Initialize()* es usada para lógica post-creación del objeto. En el caso del TEGE para poner un ejemplo, tenemos el objeto *Deck* que contiene las cartas que se dan como premio al jugador por sus conquistas. El objeto es definido y creado como variable de la clase principal pero hay más lógica inherente a un mazo, como la creación de las cartas y el mezclado de las mismas. Es por eso que dentro de *Initialize()* encontramos *deck.Create()* y *deck.Shuffle()*.

2. **Load():** al estar fuera del Loop esta sentencia se ejecuta una sola vez. En esta función se encuentra la lógica que carga los elementos gráficos a los objetos. Cada objeto que así lo requiera por tener una representación en pantalla va a contener un atributo que sea una imagen o una figura 3D. Como estos métodos que toman archivos del disco se definen en la clase principal, en vez de mandarlos a cada objeto, se centraliza este procedimiento bajo esta función *Load()*. También acá se cargan los archivos de fuentes (Fonts).

3. **Update():** una vez dentro del Loop comienza la lógica y el juego en sí. Esta función describe todo lo que es la lógica del juego, el gameplay, los objetivos, las reglas, posibilidades y los mecanismos del mismo, lo que se denomina en la jerga el *CoreEngine (núcleo).* Aparte de la lógica del juego tenemos el comportamiento de los objetos, es durante esta llamada que el estado de los mismos puede cambiar, por ejecutar una acción recursiva (como el caso de los planetas que giran constantemente) o por ejecutarse en un momento en particular (como el girar de los dados). Es aquí donde también se resuelve la IA (inteligencia artificial) de lo objetos que posean alguna.

4. **UpdateInput():** en esencia es un método con el mismo objetivo que el *Update()* pero se suele separar para ordenar su única diferencia. Mientras que *Update()* se encarga del comportamiento automático del objeto, *UpdateInput()* se encarga de la modificación del comportamiento pero causado por el usuario. Es aquí donde toda las condiciones van a ser del estilo *If(Mouse.clicked())*, *while(Keyboard.LEFT)* o *switch(Joystick.button)*, como para poner algunos ejemplos.

5. **Render():** este método, que obviamente va dentro del Loop, contiene el código relativo a la representación en pantalla de los objetos que así lo requieran. Basándose en las coordenadas de los objetos y en el tipo de elemento que sea (textura 2D o modelo 3D), los mandará al dispositivo principal de salida, llamado en algunos casos ViewPort. Un ViewPort puede ser definido como la “cámara” del juego también.

Lo importante de esta estructura es que para una estructura general simple los objetos pueden seguir esta línea, si todo esta programado dentro de cada objeto la estructura del programa se hace muy fácil de seguir. Supongamos que tenemos la clase *Jugador* y la clase *Enemigo*, si cada una de está programada respetando estos métodos (salvo el *UpdateInput()* que no se aplica a los objetos), el juego tendrá una estructura como se verá a continuación.

```
Game()
{
    Initialize() {
      Jugador.Initialize();
      Enemigo.Initialize();
    }

    Load() {
      Jugador.Load();
      Enemigo.Load();
    }

    Update() {
      Jugador.Update();
      Enemigo.Update();
     }

    UpdateInput() {
    }

    Render() {
      Jugador.Render();
      Enemigo.Render();
    }

    Unload() {
      Jugador.Unload();
      Enemigo.Unload();
    }
}
```

5.2. Gráficos 2D

Antes de observar como se utilizan las imágenes tenemos que saber como están compuestas y ver un poco de su evolución. A pesar de que todas son imágenes, se las suele llamar distinto en el ambiente de la programación de videojuegos dependiendo de la función que cumplan. Es por eso que luego se verá en este apartado los Tiles, Sprites y Texturas.

5.2.1. Profundidad del Color

La cantidad de bits determina la profundidad del color o, para entenderlo mejor, la cantidad de colores disponibles para cada píxel de la pantalla.

- **1 bit:** también denominado monocromo, cada píxel puede contener 0 o 1 como valor, indicando negro o blanco.

Monocromo

- **8 bits:** a partir de los 8-bits surgen dos tipos de formatos para las imágenes. La primera es la que contiene aparte de la imagen en sí, una paleta de colores. Entonces cada píxel utiliza sus 8-bits como una dirección de entrada en la paleta que le determina su color. 8-bits dan la posibilidad de 256 colores aunque la paleta puede ser de 16 o mas bits dando una cantidad de más de 65,536 colores. El hardware de video también posee una paleta de colores, que es actualizada con la paleta de la o las imágenes. Pero esto era un problema ya que si la tarjetas gráficas es de 8-bits significa que una imagen reemplaza esa paleta ya que son del mismo tamaño. El verdadero problema es cuando se necesita renderizar más de una imagen que poseen paletas diferentes, el dispositivo gráfico debe optar por usar una de las dos en cuyo caso es probable que una de las imágenes pierda fidelidad o unir y mezclar las dos paletas en donde se espera que las ambas sufran cambios pero menores. El segundo tipo de formato es el RGB (red green blue) en donde el color resultante de cada píxel es la suma de distintos gradientes entre rojo, verde y azul. En 8-bits la distribución es: 3 bits para el rojo, 3 bits para el verde y 2 para el azul. Este formato no usa una paleta predeterminada y se lo denomina Truecolor (color verdadero).

8-bits

- **16 bits:** en 16-bits tenemos la posibilidad de 65,536 colores. La distribución de los mismos es: 5 bits para el rojo, 6 para el verde y 5 para el azul. A las imágenes de 16-bits se las denomina Highcolor. Como regla general y como se puede observar, la distribución de los bits trata de ser igualitaria para los 3 canales o colores del RGB. Cuando sobra un bit como en este caso, se le asigna al canal verde ya que el ojo humano distingue con más facilidad distintos gradientes de este color y por eso vale la pena usar este bit (que agrega algunos tonos más) para el verde. En caso de que

sobren 2 bits el segundo se le aplicara al rojo porque es el que le sigue por la misma razón. El azul es el color que el ojo humano distingue menos sus cambios. Este formato de 16-bits se puede trasformar en 15-bits en donde el bit libre es usado para el canal Alpha de transparencia. Claro esta que al ser un solo bit el único efecto que podemos lograr es un píxel opaco o transparente, en sus extremos.

16-bits

- **24 bits:** los colores que podemos obtener con 24-bits son 16,777,216. Se utilizan 8 bits (1 byte) para cada canal. Como se ve en la imagen a continuación no hay casi diferencia entre la de 16-bits, estos es porque el ojo humano tiene una limitación sobre los colores que es capaz de distinguir (se estima que alrededor de 10 millones). Para que realmente haya una notable diferencia entre una imagen de 16 y 24 bits, la última debe ser de gran tamaño y preferentemente reflejar algo de la realidad.

24-bits

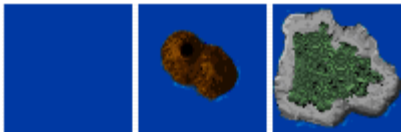
- **32 bits:** en calidad una imagen de 32 bits es igual a la de 24-bits, el byte extra se utiliza para Alpha blending. A diferencia de una imagen de 15-bits que sólo posee un bit para transparencia; con un byte disponible los efectos abarcan un rango 256 posibilidades de tonos de transparencia, dejando ver más uno u otro de los colores superpuestos o mezclando ambos. Es por esto que se puede decir que una imagen de 32-bits sin canal Alpha es una imagen se pasa a 24-bits y viceversa. Ya que los espacios en disco están mas evolucionados y no hay problemas de falta del mismo, se utilizan directamente imágenes de 32-bits aunque no sea utilizado el cuarto canal ya que a nivel de procesamiento, 24 bits se manipulan con multiplicaciones de 3 que requieren más tiempo mientras que manejar 32 bits utiliza multiplicaciones de 4 que se pueden lograr más fácilmente solamente invirtiendo los bits.

5.2.2. Tiles

Ahora que sabemos como están compuestas las imágenes, veamos el juego 1942 como ejemplo gráfico de cómo nos pueden ayudar los Tiles.

1942

El jugador sale de un portaaviones para recorrer un mapa de agua y tierra hasta llegar a otro portaaviones del otro lado del mapa, el jugador está aproximadamente dos minutos por nivel. Si queremos crear una versión de mejor calidad pensaremos en una versión de para la resolución 1024x768. Para que el jugador esté dos minutos en el aire tendría que recorrer varias de estas pantallas, nuestra imagen del mapa podría tener tranquilamente algunos metros de largo. Si tarda 10 segundos en recorrer una pantalla, para que dure 120 segundos, tendría que 768 x 12 = 9216 píxeles de alto, tendríamos una imagen de 1024x9216. Ahora bien, como necesitamos la mas alta calidad cada píxel es de 32-bits, o sea 1024 x 9216 x 32 = 301989888 bits que equivalen a 3 megas. Un fondo con una imagen de 3 megas actualizándose continuamente es impracticable, requeriría demasiado poder de procesamiento para el tipo de juego que estamos realizando. Por esta razón y para este tipo de juegos se utiliza la lógica de los Tiles (azulejos en castellano). Ya que las imágenes son iguales no es necesario tener un mapa tan grande siendo renderizado al mismo tiempo todo el tiempo, si sectores del mapa son iguales no vale la pena guardar toda esa información redundante y si el mapa es más grande que la pantalla significa que se renderizar porciones que no se ven. Cada tile es un fragmento de imagen que unidos a otros tiles, forman un tilemap que es una imagen más grande y completa. La imagen de arriba se puede compuesta solo por los siguientes elementos: Agua, Roca e Isla.



Tiles

Transformamos nuestro mapa completo de 1024x9216 píxeles en 3 imágenes de 60x60 que suman un total de 180x60. Igualmente claro esta que esto debe ser llevado a pantalla. Si pre-armamos todo el mapa una sola vez habremos ahorrado en espacio pero no en

procesamiento. Como no es necesario todo el mapa ya que solo se ve una porción lo que se debe crear es justamente esa porción que se va viendo al mismo tiempo que liberamos la porción por la que el jugador ya pasó. Vamos a tener como base obviamente una imagen de 1024x768 en pantalla, pero agregaremos una línea de tiles que se van a pre-cargar antes de llegar a la pantalla y dejaremos que una línea de tiles salga de la pantalla antes de eliminarla. Para entender esto lo ejemplificaremos en el siguiente gráfico.

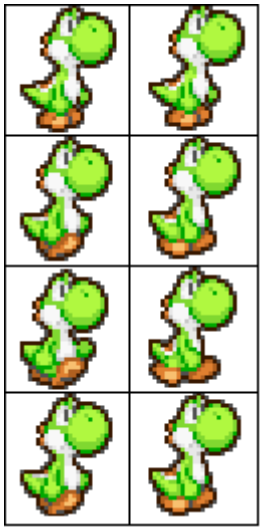
Ahora tenemos una imagen de 1024x768 con 3 líneas más de 1024x60. Una inicial que se pre-carga, una final que al irse de la pantalla se elimina, y una auxiliar que generalmente se usa para que tanto la carga y el borrado sean invisibles al usuario. Sin esta línea el cambio se hace con un píxel de diferencia y con la velocidad que puede tener el scroll (vertical en este caso) se pueden llegar a ver líneas negras arriba en la pantalla. Hemos pasado de ocupar 3 megas de espacio en disco y 3 megas en memoria, a ocupar, 180x60 => 10 kilobytes de espacio con un tilemap y 1024x768 + 1024*60 * 3 => menos de 1 mega de memoria.

5.2.3. Sprites

Sprite es la terminología básica para una imagen 2D que representa un objeto en pantalla generalmente asociada a un actor, personaje, enemigo u objeto; un fondo de pantalla, la GUI para el usuario no reciben esta definición, básicamente porque un sprite puede ir animado, mientras que un fondo de pantalla generalmente no. Anteriormente como un proceso de optimación la imagen era trasformada en otra representación lógica para ser enviada a pantalla, esto era lo que se denominaba Sprite. Como la performance de las computadoras mejoró notablemente ya este proceso no es necesario y el termino Sprite quedo para las imágenes en sí.

Ejemplo de un Sprite de una nave

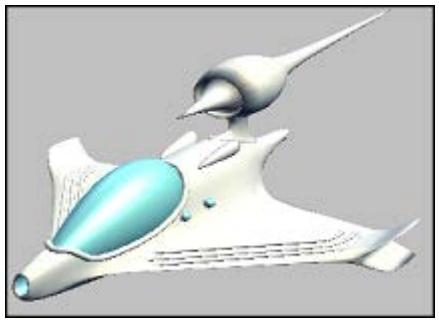
Para animar un sprite el concepto es el mismo que el de una película en un cine, la animación viene dada por una secuencia de imágenes, que a una velocidad de 60 FPS dan la sensación de animación. Toda la secuencia se recomienda que este en un mismo archivo, por un tema de ordenamiento. Luego en el código se hará una función que corte en rectángulos iguales (que así deben estar dibujados) las imágenes pertenecientes a la animación, se guardará en un arreglo de imágenes y es el que se utilizará en el juego recorriéndolo continuamente.



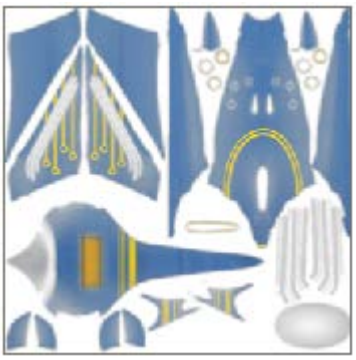
Ejemplo de la imagen para Secuencia de Sprites

5.2.4. Texturas

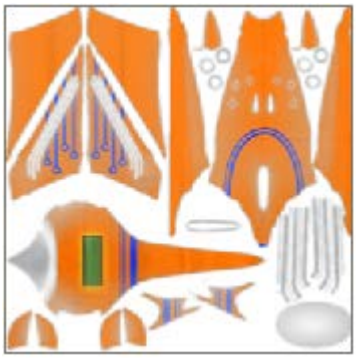
Este término está relacionado a las imágenes que se aplican sobre los modelos 3D para darles color y motivos. Cuando se crea un modelo 3D, el mismo no esta “pintado”, se puede pintar directamente sobre el modelo (si la herramienta lo permite) que no es lo recomendado ya que es más difícil de modificar y aparte porque se busca la reutilización del modelo, de la otra forma podemos tener un mismo modelo con distintas texturas y nos sirve para representar dos objetos distintos en pantalla. En el ejemplo a continuación, basándose en un mismo modelo pero con distintas texturas, obtenemos dos objetos que podríamos utilizar para distinguir las naves de dos jugadores. En el programa de edición 3D lo único que se pintó es el vidrio de la cabina (y otros detalles pequeños) ya que son iguales para todos.



Modelo Base sin textura

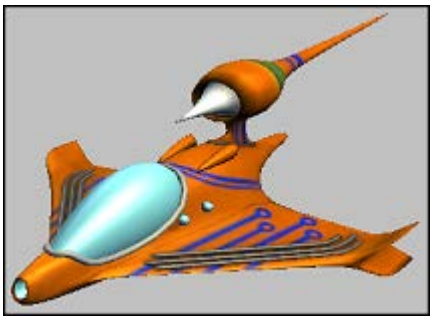


Textura 1



Textura 2

Objeto final con Textura 1 aplicada



Objeto final con Textura 2 aplicada

5.3. Gráficos 3D

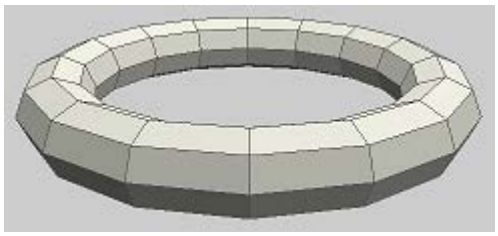
5.3.1. Meshes

Los modelos 3D están compuestos por lo que se llama Mesh. La traducción de Mesh es entretejido, grilla o red y como se verá los modelos 3D están formados por esta red poligonal. La red o Mesh está formada por:

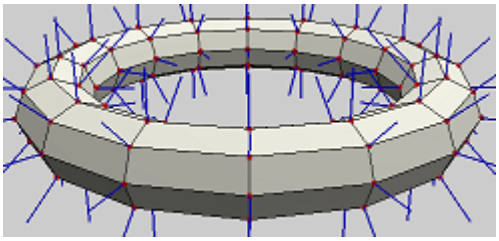
- **Vertex (vértices):** es un punto en el espacio que contiene información sobre el color y coordenadas de la textura aplicada (si hay). Cada vértice contiene una vector que es su normal.

- **Edge (bordes):** un borde esta formado por dos vértices. Cada borde también contiene un vector normal que va a ser perpendicular a si misma.

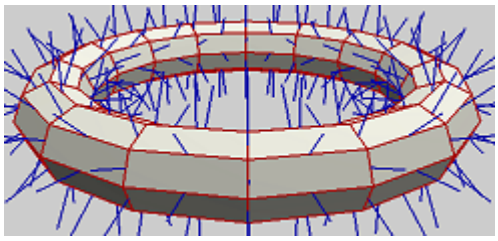
- **Face (caras):** una cara esta formada por los bordes que la rodean y que se cierran. Una cara siempre va a tener una forma triangular o cuadrangular y forma un plano. El vector normal va a ser perpendicular al plano que forma.



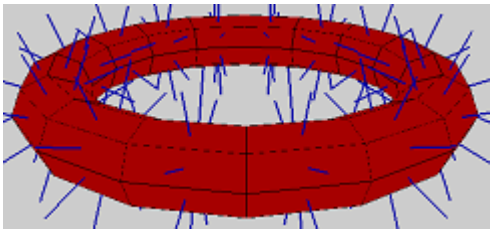
Modelo 3D de un aro



Vértices con su normal



Edges con su normal



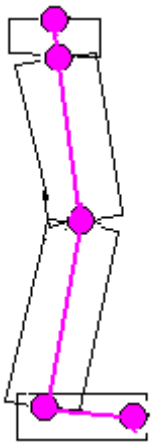
Faces con su normal

En su forma más básica un modelo 3D esta compuesto por vértices pero como no es suficiente y para acelerar el renderizado se agrupan en Edges y luego en Faces. Como se ve todos estos elementos tienen información adicional sobre, color, textura, luz, brillo y transparencia, en especial, la normal de cada uno se utiliza para los efectos de la luz.

5.3.2. Bones

Un bone o hueso como bien lo indica la palabra puede formar parte del esqueleto de un mesh. Los bones se utilizan para animar modelos 3D (que obviamente están formados por más de un mesh) y facilitan el hecho de que si movemos un mesh dentro de un modelo, no tenemos que mover manualmente todos los demás. Los bones forman parte del mesh y se comunican con los bones de otros meshes y cuando uno de estos se mueve, por la unión que tienen entre sí, los demás meshes se moverán acordemente. Imaginemos el ejemplo de una pierna, podemos tener 4 Meshes que serían la cadera, el muslo, la pierna (como parte del miembro inferior) y el pie. Cada Mesh tendría su propio bone, y estarían unidos cadera con muslo, muslo con pierna y pierna con pie. El movimiento del pie ocasiona una reacción en cadena entre los otros Meshes. En el caso de un videojuego el uso de bones evita la necesidad de programar el

comportamiento de cada Mesh del modelo y en caso de animación computarizada evita tener que dibujar cada Mesh a una nueva posición por el movimiento de uno.



Bones uniendo Meshes

5.4. Colisiones

Ahora que conocemos los elementos básicos que conforman un juego, lo más importante es saber como interactúan entre sí. La acción más común entre objetos son las colisiones, Pac-Man comiendo fantasmas, Mario saltando arriba de tortugas y proyectiles destruyendo enemigos, todo esto es posible cuando se piensa que todos estos actores u objetos están colisionando unos con otros.

5.4.1. 2D

Las colisiones en el plano se pueden lograr de dos formas:

- **Bounding Box:** para la primera se sabe que un sprite, sea la imagen que sea, es siempre por definición un rectángulo, de ahí el término de esta técnica que chequea los rectángulos que contienen la imagen. Una función CheckCollision() que reciba como parámetro dos objetos que contengan la información de su posición y la información (ancho y alto) sobre la imagen que los representa en pantalla puede usar el siguiente algoritmo:

```
If (rectA.X + rectA.Width > rectB.X - rectB.Width &&
   rectA.X - rectA.Width < rectB.X + rectB.Width &&
   rectA.Y - rectA.Height > rectB.Y + rectB.Height &&
   rectA.Y + rectA.Height < rectB.Y - rectB.Height)
Then
    Collision;
```

Bounding Box con colisión

Como se ve *Bounding Box* puede servir para la mayoría de los juegos y/o situaciones. En el caso de que nuestros sprites se asemejen más a círculos se puede utilizar lo que se llama *Bounding Sphere* que es en sus fundamentos es el mismo chequeo de distancia, pero en vez de usar el ancho y alto que posee un rectángulo, se utiliza el radio del círculo como medida principal.

- **Pixel per Pixel:** a veces es necesario por los sprites que utilizamos un algoritmo más preciso para la detección de colisiones. La técnica *Pixel per Pixel colision* es un algoritmo más lento que el *Bounding Box* pero ofrece resultados perfectos. El algoritmo chequea el rectángulo formado por la colisión entre los sprites, y revisa para cada píxel de ese rectángulo, el equivalente en posición de cada una de las imágenes para cada píxel. O sea toma el primer píxel del rectángulo en colisión luego toma de ambos sprites el píxel que correspondería a esa ubicación y si ambos píxeles contienen color considera que hay colisión. Los otros resultados posibles son si uno de los dos píxeles no tienen color o tienen el byte o bit Alpha prendido; u obviamente si ambos son píxeles con información en el canal Alpha.

Bounding Box detecta una colision en este caso ya que los rectángulos del avion y de la bala se superponen.

Pixel per Pixel no detecta un colision ya que aunque las imagenes esten superpuestas no hay dos pixeles NO vacios en contacto.

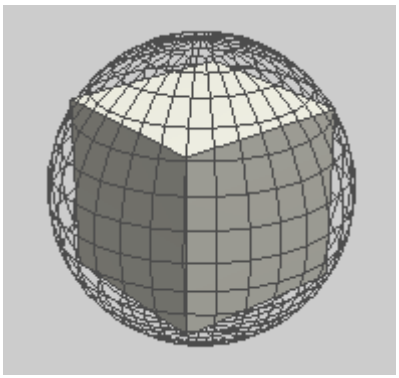
En este caso Pixel per Pixel si detecta una colision que es mucho mas precisa que la de Bounding Box.

5.4.2. 3D

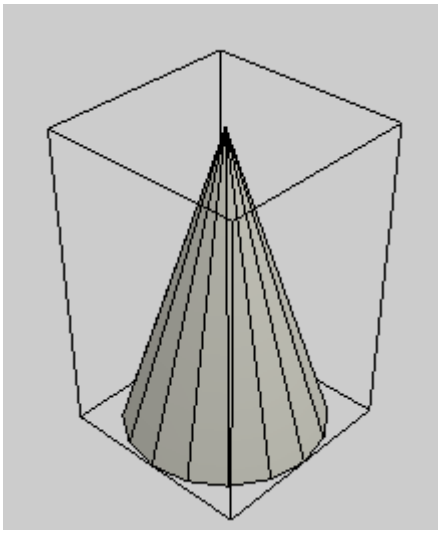
Las colisiones en 3D usan los mismos conceptos que las colisiones en 2D, incluso algunas técnicas reciben el mismo nombre:

- **Bounding Sphere:** en el caso de dos dimensiones y por la estructura que tiene un sprite, la misma ya esta contenida en un cuadrado (por saber el ancho y alto de la

imagen), de ahí el nombre de la técnica más utilizada, *Bounding Box*. En cambio en el mundo 3D un modelo no esta encapsulado dentro de otro objeto o marco, por ello el *Bounding Sphere* o *Bounding Box* debe ser creado por nosotros. La primera, la esfera, que es la más sencilla, toma como parámetro la distancia entre el centro del modelo y el vértice más lejano, y crea una esfera que cubre el modelo utilizando ese valor como su radio. Como cada objeto tiene una *Bounding Sphere* asociada, los chequeos se hacen con las mismas simplemente detectando las distancias entre ellas y utilizando como parámetro principal el radio. Si la distancia entre los puntos centrales de cada modelo es inferior a la suma de ambos radios, hay colisión. Cuando la forma del modelo es particularmente más extensa sobre uno de sus ejes, o sea por ejemplo: mucho más alta que ancha, o mas ancha que profunda, es cuando se utiliza *Bounding Box*. Para simplificar las operaciones matemáticas posteriores el *Bounding Box* es ubicado obviamente también con centro en el modelo pero perpendicular a los eje.

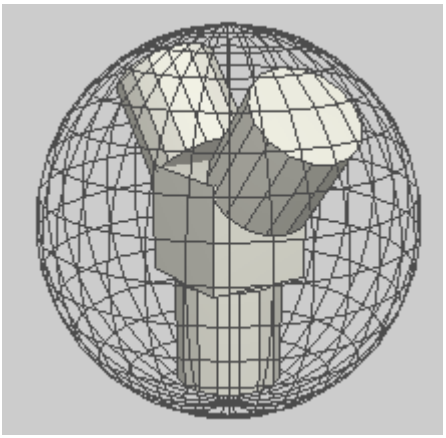


Bounding Sphere

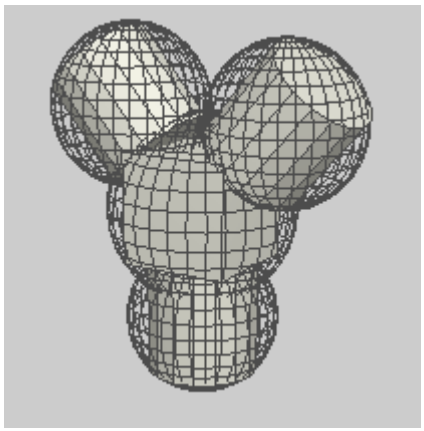


Bounding Box

Se aclara que esta esfera o caja no tienen ninguna representación, esto es sólo un caso para facilitar la visualización, en realidad son puramente lógica matemática que nos sirven para las colisiones. Claro, hay modelos que no se ajustan a ninguno de estos parámetros, o sea que sean uniformes en sus dimensiones o tengan sólo una de sus dimensiones en distinta proporción, esto se da con modelos más complejos compuestos por muchos Meshes, en este caso lo que podemos hacer es asignar a cada Mesh del modelo su propio *Bounding Sphere*, se requiere más procesamiento pero si se requiere algo más de precisión es necesario.

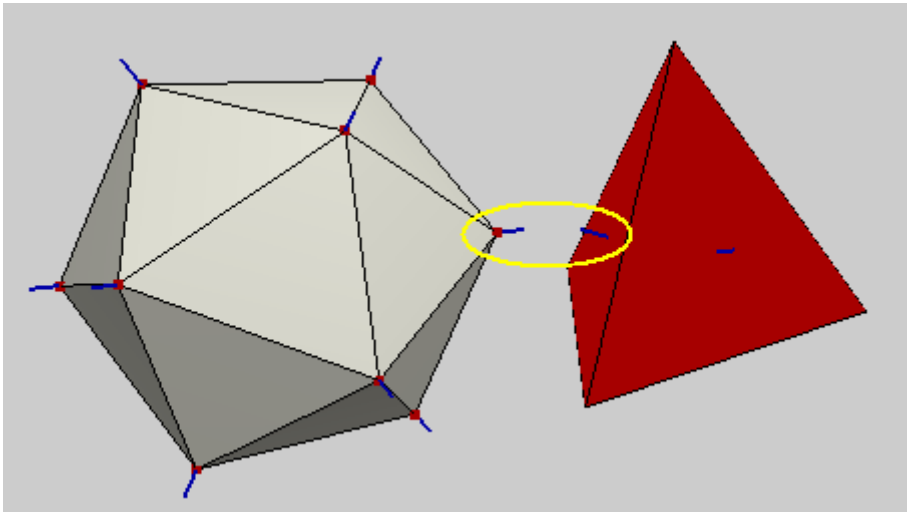


Un Bounding Sphere único



Multiples Bounding Spheres

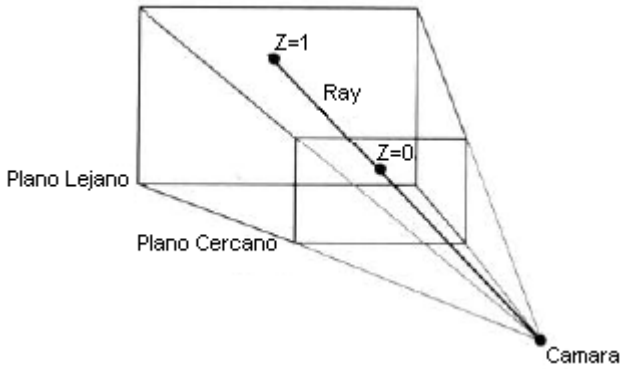
- **Punto a Punto:** En caso de que los modelos tengan formas realmente poco comunes podemos utilizar una comprobación sobre los puntos que contienen al modelo. No se habla de píxeles ya que no están compuestos por los mismo, ni siquiera existe en el plano la comprobación de un bit o byte Alpha o similar. La comprobación de absolutamente todos los puntos de un modelo con los del otro no es una práctica recomendable. Por lo que se sigue en cambio el siguiente acercamiento. Se debe comprobar todas las Faces de un modelo con todos los vértices del otro y de acuerdo a sus normales determinar que par se encuentra más cerca. Una vez encontrado esto se debe chequear si el vértice evaluado atraviesa el plano formado por la Face del otro. El par que resulto del algoritmo se puede guardar para ser el primer par en ser evaluado al siguiente ciclo y si no hubo cambios ahorrar tiempo en esta comprobación.



Un Vértice de un modelo y una Face del otro

5.4.3. Híbridas

Por como se vio hasta ahora parecería imposible que un objeto 2D interactúe con uno que se encuentra en el espacio 3D. Para lograr esto se hace pleno uso de matemática trigonométrica que nos permitirá transportar una coordenada de dos dimensiones a una de tres y viceversa. Para explicarlo conceptualmente y mantener los cálculos matemáticos al mínimo debemos decir que básicamente a una coordenada 2D le asociamos un Ray (rayo) que se introduce en el espacio 3D. Para lograr esto vamos a entender a la pantalla como un cono que tiene profundidad y tiene dos planos, el más cercando donde estaría el mundo 2D y el plano lejano donde se encuentran los modelos. Lo primero que vamos a hacer es trasformar las coordenadas del Mouse (por ejemplo, es el caso típico) en dos vectores de tres dimensiones (uno indicando el plano cercano y otro el lejano), para ellos, respetamos las coordenadas X e Y y para Z colocaremos, 0 para plano cercano y 1 para el lejano. Ambos vectores se proyectan hacia el espacio 3D para encontrar sus posiciones 3D reales. Se crea un tercer vector tridimensional que es la resta entre los otros y sacaremos su normal para saber la dirección. Teniendo como referencia el punto "cercano" y una dirección que va a ser hacia el fondo, hemos creado el rayo. Cuando movamos el Mouse existirá un rayo lógico que colisionará con los modelos ubicados en el espacio, dando la sensación de que en realidad es el Mouse el que lo hace.

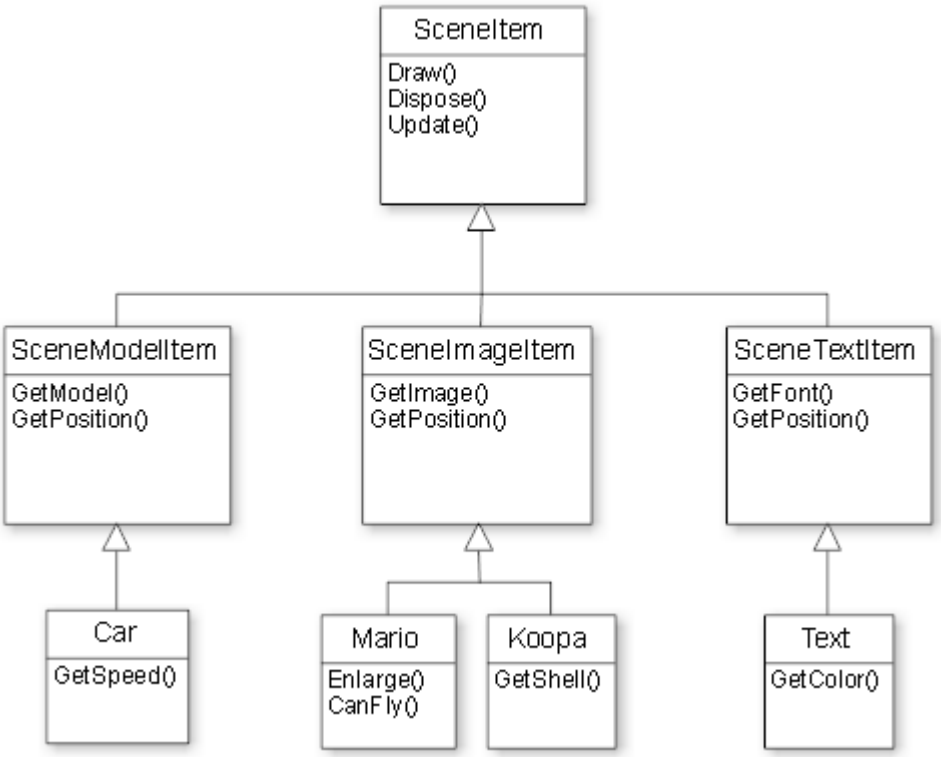


Ray definido por dos puntos en distintos planos

5.5. Estructura

Ahora que ya tenemos noción de los elementos y como interactúan entre sí lo recomendable para tener un código fácil de leer, modificar y mantener es encontrar una estructura que nos servirá a modo de Framework para la programación de los objetos. Un juego es claramente una aplicación visual y como tal uno de lo núcleos más importantes recae en el *Engine Gráfico* (que se verá en detalle en el siguiente apartado). El *Engine Gráfico* trabaja con lo que se denomina *Content Pipeline* que posee todos los elementos que requieren estar en pantalla y por ende ser renderizados. Si nuestro código no sigue cierta lógica cada objeto tendrá su propia forma de actualizarse, de dibujarse y habría mucho código duplicado y poco mantenible. Por esa razón una actividad como el renderizado debe estar centralizada en el

Engine Gráfico y los objetos agrupados en un *Content Pipeline* que los pueda manipular. Cada objeto que sabemos que tiene que ser dibujado en pantalla, digamos una nave, un personaje o simple texto es un objeto que se denomina de escena, *SceneItem.* Ahora bien, todos estos objetos no son a nivel estructura iguales, tenemos modelos 3D con tres coordenadas y matrices de ubicación, sprites con posición en dos coordenadas con efectos posibles como flip y también podemos tener texto que no tiene ni modelo ni imagen pero si usa fuentes TypeFonts. Entonces podemos crear subcategorías que heredarán de *SceneItem* como *SceneModelItem*, *SceneImageItem* y *SceneTextItem* para este ejemplo. La clase base *SceneItem* tendrá en primera instancia métodos básicos como *Draw()* para dibujar, *Dispose()* para indicar que no se desea dibujar más el objeto y *Update()* que si se recuerda contiene lógica que cambia el estado del objeto. El método *Draw()* no será el encargado de dibujar el objeto sino que lo pondrá en el *Content Pipeline* y *Dispose()* lo sacará del mismo. En las subclases podemos implementar cosas más específicas para cada una de ellas, en el caso de *SceneModelItem* tendremos *GetModel()* y *GetPosition()* que devuelve un Vector de tres coordenadas, en *SceneImageItem* en cambio estaría *GetImage()* y un *GetPosition()* que devuelve un Vector de dos coordenadas y en *SceneTextItem* habría un *GetFont()* por ejemplo. Cuando creemos una clase que sabemos que debe ser dibujada la haremos heredar de una de las subclases, con código mínimo más un *Engine Grafico* acorde, con crear el objeto, inicializarlo y llamar al método *Draw()* ya lo estaríamos viendo en pantalla porque el código de renderizado esta en el *Engine Grafico* así como también funciones adicionales que mejoran esa función y la hacen transparente al objeto en sí.

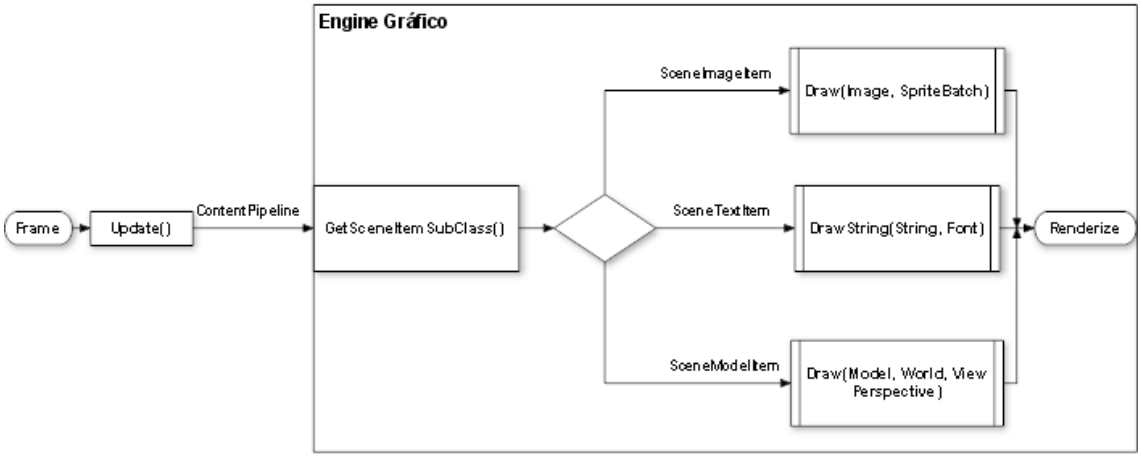


Ejemplo de una posible estructura

5.6. Engine Gráfico

Lo que se denomina Motor Gráfico se debe ver como el elemento que procesa el *Content Pipeline* y centraliza todas la operaciones de dibujado decidiendo "que" y "como" hacerlo. Este elemento es uno de los más importantes dentro de un juego y es tanto crítico como complejo de lograr. Para no extender demasiado este apartado se darán sólo algunos conceptos básicos sobre su función y estructura. El *Content Pipeline* que en estructura puede ser una lista (así también como un árbol u otra estructura de datos) contiene los objetos *SceneItem* que son entregados al *Engine Gráfico*, éste los evalúa y dependiendo de la subclase a la que pertenezcan sabe como tratarlos en relación a la renderización adecuada para cada uno de ellos. Para dibujar imágenes a las mismas se las procesa con lo que se puede llamar un Batch de Sprites, que toma estas imágenes y las va ubicando en el buffer del dispositivo de salida a lo largo de la ejecución del Frame y al final de este dibuja lo que se encuentra en este *FrameBuffer*. Para el caso de lo que es texto también es llevado al FrameBuffer, pero la función que lo ubica ahí en vez de ser un simple *Draw(),* recibe el nombre de *DrawString()* que tiene un parámetro que es la fuente. El caso de 3D es un poco más complicado, son necesarias tres matrices que se definen como la matriz *World* que contiene la información sobre la posición y rotación de un objeto, la matriz *View* que indica de donde y hacia donde esta la línea de visión y la matriz *Perspective* que define el mínimo y máximo de nuestro alcance de visión. Mientras que la matriz *World* es un atributo propio de cada modelo,

View y *Perspective* están sujetas a parámetros de la cámara y esto es así para mantener una coherencia en cuanto a donde se dibujan los modelos. El *Engine Gráfico* va a estar encargado de actualizar estas dos últimas matrices de acuerdo a lo que tome como datos de una clase *Camara* que por estas razones es útil crear y definir.



El Engine Gráfico

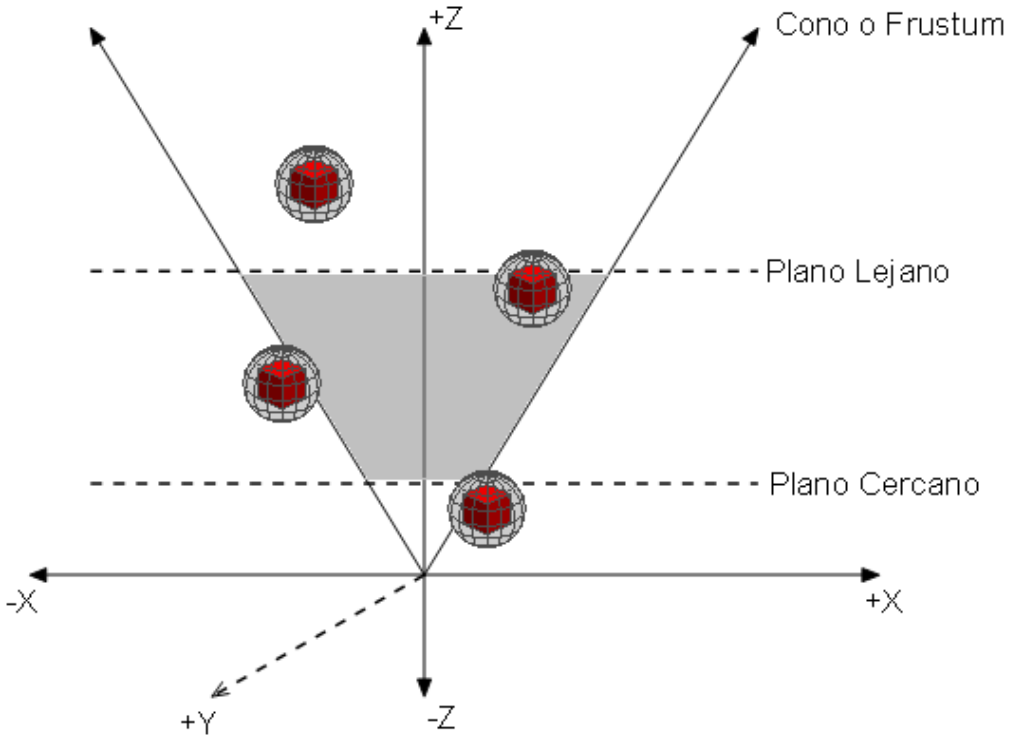
El gráfico de arriba muestra de forma simplificada como por cada Frame del juego se realiza la operación de *Update()* (podríamos haber incluido *UpdateInput()* también) que modifica nuestro *Content Pipeline* agregando o sacando elementos y como esta lista actualizada es procesada por el motor gráfico que dependiendo del tipo de objeto los renderiza de forma distinta terminando así con el procesamiento de ese Frame.

5.7. Engine Gráfico Avanzado

Un motor Gráfico Avanzado no debe procesar el *Content Pipeline* en su totalidad sino que debe depurarlo antes para reducir el número de objetos en el mismo y acelerar el renderizado y por ende el procesamiento en general obteniendo mejor velocidad o *Framerate*. Esta optimización no se aplica sobre el contenido 2D, se espera más que nada que dependiendo del estilo de juego 2D la lógica sea coherente en cuanto a lo que se dibuja en pantalla, o sea no dibujar imágenes que estén muy lejos de la pantalla que se esta viendo en este momento. Lo que si se puede aplicar en el Engine Gráfico que ayuda más a los juegos 2D es la utilización de un algoritmo que actualice sólo la parte de la pantalla que sufrió un cambio en vez del típico “borrar y dibujar todo devuelta”. Esto se denomina *Dirty Area Computation* y se encarga de determinar el área “sucia” del juego, crear un rectángulo con esa área y en vez de computar un *Draw()* general lo a través de un *Draw(x,y,w,z)* que trabaje sólo sobre el área determinada. En los juegos 3D en donde generalmente se ve un mundo entero, casi la totalidad de la pantalla cambia frame a frame por lo tanto no resulta tan útil esta técnica como sí lo puede ser en una escena 2D que es más estática. Un mundo 3D es infinitamente más grande

que lo que podría ser una imagen (recuerden el apartado sobre Tiles y la imagen de 3MB) por lo que es más importante y totalmente necesario reducir el numero de objetos del *Content Pipeline* con la premisa “no dibujar lo que no se ve”. Para lograr esto existen diversas técnicas que se explican a continuación:

- **Culling:** dentro del mundo 3D que nos ubiquemos sólo veremos lo que esta delante de nosotros. *Culling* entonces basándose en la matriz de *View* y *Perspective* crea un cono de visión (denominado *Frustrum*) y compara con la matriz World de cada objeto cuales de ellos se encuentran dentro de este cono. Para ellos se hace una verificación de posición no con el modelo en sí sino con el *Bounding Sphere* o *Box* asociado a cada uno de ellos. Sólo aquellos objetos que estén dentro de esta área o aquellos cuyo *Bounding Sphere* o *Box* colisione (incluso aunque la posición este fuera) con el *Frustum* serán renderizados.

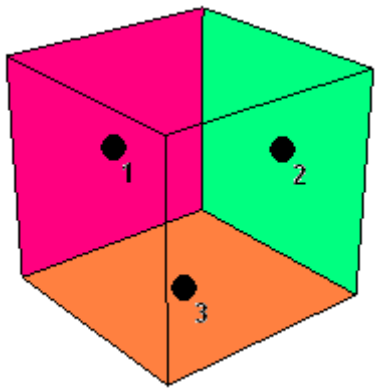


Ejemplo de los límites que puede tener el Frustum

Para el ejemplo anterior se puede observar (de arriba hacia abajo) un cubo claramente fuera del área y por ende no renderizado o sea removido del *Content Pipeline*, un cubo con posición adentro del cono y que entonces va a ser renderizado, un cubo con posición fuera del área pero con su *Bounding Box* colisionando o sea se va a mantener dentro del *Content Pipeline* y finalmente un cubo con su *Bounding Box* colisionando que es un ejemplo de una falla que puede ocasionarse dentro del *Culling*

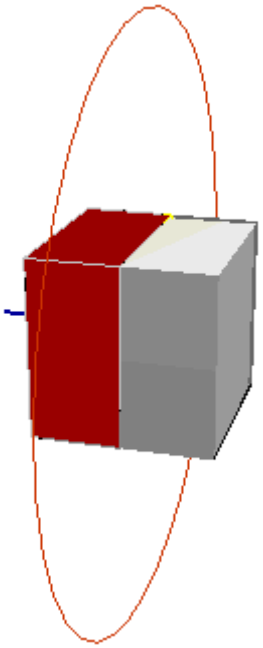
ayudado por una mala definición de los que es el límite del plano cercano. Falla que como se verá puede ser solucionada con *Clipping*.

- **Back Face Culling:** la lógica detrás del *Back Face Culling* se puede explicar de la siguiente manera. Un cubo tiene seis lados pero es imposible ver más de tres al mismo tiempo en cuyo caso hay tres lados que no serían necesarios dibujar. Hay que recordar que los modelos usados en videojuegos se denominan *Básicos* a diferencia de los denominados *Sólidos* usados por ejemplo para simulaciones. Esto significa que cada lado de un polígono representa una cáscara de espesor infinitamente finita. Una pared es un juego no es un objeto sólido, es solo una imagen de una profundidad tan delgada que vista de costado no existiría y si nos ubicamos exactamente en su posición y nos movemos una fracción de coordenada mas hacia delante la habremos pasado. Teniendo en cuenta esto se puede entender que las caras que no se ven de un polígono son muy poco importantes. Basándose en el Vector Normal de cada polígono, aquellos que tengan una dirección opuesta a donde se encuentra el "ojo" o *POV* (*Point of View)* serán eliminados. Nótese que no se elimino el modelo sino que se trasformo en un conjunto de polígonos de los cuales se evalúo que no eran todos necesarios.



Back Face Culling elimina las caras 1, 2 y 3

- **Clipping:** esta técnica es similar *Back Face Culling* porque traduce un modelo en un conjunto de polígonos y elimina aquellos que no son necesarios. Pero esta evaluación trata de determinar de aquellos modelos que están parte dentro y parte fuera del *Frustum* (lo mismo para sus polígonos). Básicamente si un modelo esta contra un borde de la pantalla y estoy viendo una mitad significa que no es necesario renderizar la otra mitad. Los vértices del modelo que estén dentro del *Frustum* son los tomados en cuenta y los polígonos que usen esos vértices serán dibujados. *Clipping* soluciona la falla que posee *Culling* para modelos que están entre el punto con coordenada Z = 0 y el plano cercano, si bien *Culling* determina que debe ser renderizado, *Clipping* determina que ninguno de sus polígonos debe ser renderizado.



Clipping suponiendo que mitad del cubo este fuera del Frustum

5.8. Engine de Sonido

Antes de comenzar con su estructura veamos de que forma podemos reproducir los sonidos dependiendo de cuantos canales de reproducción utilicemos. La utilización de varios parlantes no significa la utilización de varios canales, pero la reproducción de varios canales requiere al menos igual cantidad de parlantes. Existen técnicas de simulación de canales, lo que se llama sonido 3D pero para este escrito las vamos a omitir:

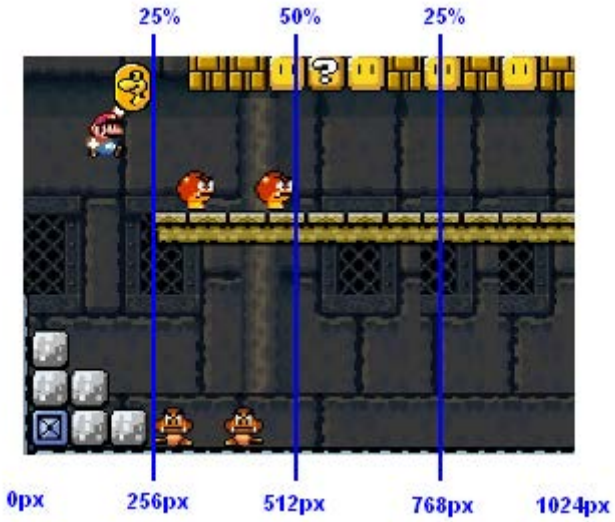
- **Mono:** significa la utilización de un canal único de sonido. Todos los sonidos salen por *todos* los parlantes (u otros dispositivos de salidas).
- **Stereo:** se utilizan dos canales, los sonidos pueden reproducirse por uno, otro o ambos. Cuando se los reproduce en ambos puede ser con la misma cantidad de decibeles (volumen) o asimétrico.
- **Surround:** misma teoría que para sonido Stereo pero utilizando de 3 a incluso 22 canales y misma cantidad de parlantes. La ubicación de parlantes ya no sólo varía hacia los costados de la persona sino que se ubican también atrás, adelante e incluso a distintas alturas.

De la misma forma que se recomienda tener todo el renderizado centralizado, con el sonido pasa lo mismo, la única diferencia es que el Engine de Sonido puede no estar tan relacionado con los objetos en sí:

1. En su forma más simple no existe absolutamente ninguna relación. La clase puede estar compuesta de métodos que describan el tipo que reproducen, el método tendrá el valor de Cue (o pista) y llamará a otro método *Play()* o similar con ese valor de pista. Dentro de la lógica del juego se creará un objeto del Engine y dentro del método *Update()* cuando se lo requiera se harán las llamadas a los métodos. Si por ejemplo Mario colisiona con una moneda, dentro de esa condición existirá una línea similar que puede ser *SoundEngine.playGetCoin().*

2. La otra forma sería que cada objeto posea lo valores de Cue's que puede llegar a utilizar y el Engine tendría un método standard posiblemente llamado *Play(Cue cueName)*. Siguiendo el ejemplo anterior ahora el objeto Coin tendría un atributo Cue con su nombre y cuando Mario colisiona con ella la línea dentro de la condición del *Update()* se podría definir de la siguiente manera *SoundEngine.Play(Coin.getCue()).*

3. La tercera forma sería para satisfacer efectos de sonido más avanzados. Para ellos estamos hablando de Stereo y Sorround solamente. Para lograr estos efectos el Engine de Sonido tiene que ser capaz de según sea el caso determinar la ubicación de los objetos en pantalla y mediante algún algoritmo definir por donde y en cuantos decibles reproducirlo.



Ejemplo de una posible función Stereo

Para el ejemplo de arriba podemos considerar una pantalla de 1024x768 píxeles. Tenemos un evento que es Mario agarrando una moneda que produce un sonido en particular. Para ser uso de una tecnología Stereo podríamos preparar el Engine de Sonido para lo siguiente. Al momento de la colisión, se le envían al Engine el objeto entero o el Cue y la posición del mismo. La posición en X de la moneda se encuentra

en el sector izquierdo que ocupa un 25% de la pantalla. Si nuestros sonidos están grabados a 10 decibeles, podríamos hacer que el sonido salga al 75% (7,5 decibeles) del volumen por el parlante izquierdo y al 25% (2,5 decibeles) por el parlante derecho. Esto crearía la efectiva sensación de que algo esta ocurriendo contra el margen izquierdo de la pantalla.

6. Software TEGE

En esta sección hablaré del producto de software en sí al que llame TEGE por el juego original pero agregándole esa última "E" para indicar "…*del Espacio"*. La teoría y marco teórico que se vino mencionando hasta el momento son para entender las decisiones que tomé en cuanto a la estructura y a los aspectos del videojuego desarrollado.

6.1. Definición

Como se dijo que iba a ser, el videojuego es un juego del género de estrategia, y para definirlo aún mejor perteneciente a la rama TBS (turn-based strategy, estrategia basado en turnos) a diferencia de los RTS (real-time strategy, estrategia en tiempo real). Juegos de este estilo como la mayoría de los juegos de mesa, se juega sobre un tablero. Para cambiar el escenario del mapa mundial elegí un enfoque más moderno y ubicar el juego en el espacio, reemplazando el mundo por el universo, los continentes por galaxias y los países por planetas. Como se sabe el juego no posee opciones multi-jugador ni jugadores controlados por la maquina (jugadores de IA).

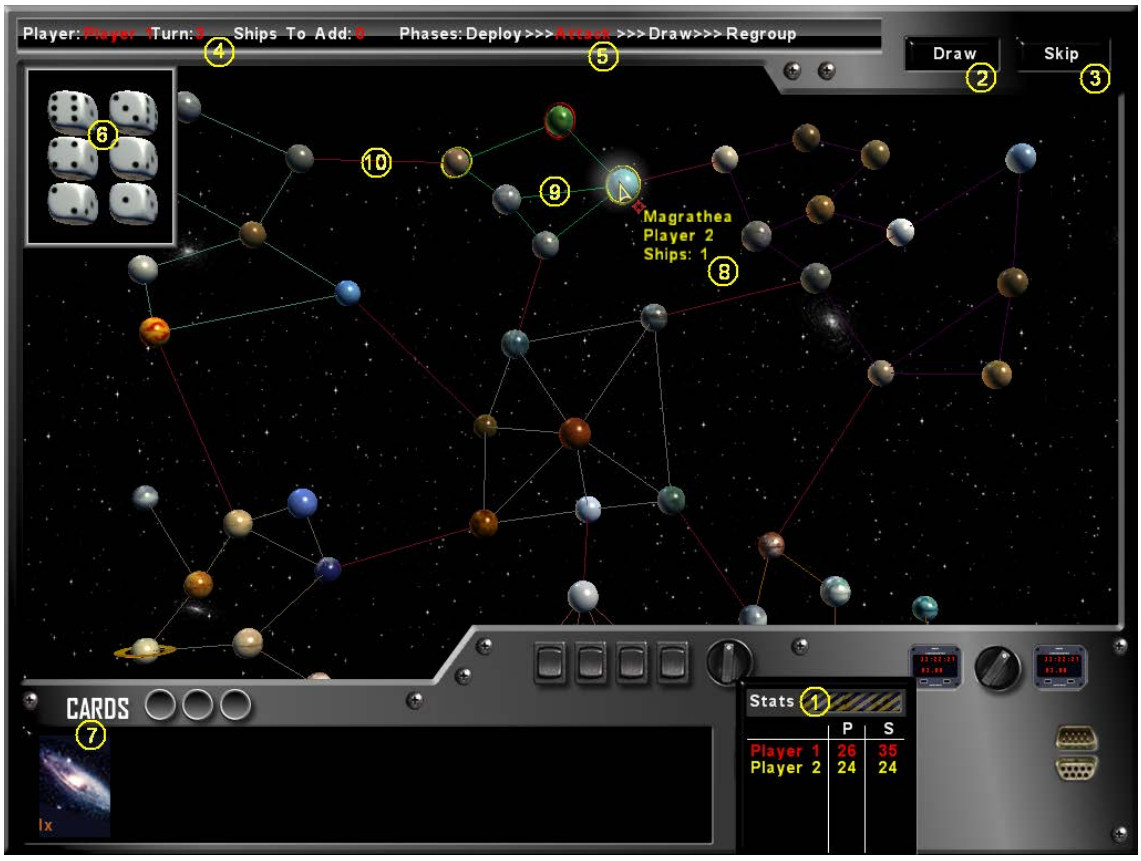
6.2. Requisitos

Para jugar a este juego es necesario satisfacer los siguientes requisitos:

- DirectX 9.0c (incluido en el CD)
- Windows XP con al menos Service Pack 2
- Placa de video que soporte al menos Pixel Shader 1.1
- .NET Framework 2.0 redistribuible (incluido en el CD)
- XNA Framework 1.0 redistribuible (incluido en el CD)

6.3. Menú Principal

Como la mayoría de los juegos, este comienza con un Menú Principal. Decidí respetar esta estructura clásica para darle al juego un tono un más profesional y que no sea considerado prototipo de prueba que sólo muestra alguna funcionalidad específica únicamente. Navegando por el menú se puede comenzar una nueva partida, ir a la parte de opciones para cambiar el volumen de la música y de los efectos, ver los créditos o salir de juego. Cuando comenzamos una nueva partida podemos seleccionar el número de jugadores. Una vez iniciado un juego podremos iremos a la pantalla del juego con los siguientes elementos que hacen a la interfaz del usuario.



1. Stats - Información sobre la cantidad de Planetas y Naves que posee cada jugador.
2. Botón de “Draw” - para robar carta durante la fase de *Draw.*
3. Botón de “Skip” - para ir pasando las fases.
4. UI – Player - Información sobre el jugador que está jugando en ese momento, el número de turno en general y la cantidad de naves disponibles que el jugador posee para ubicar en sus planetas durante la fase de *Deploy*.
5. UI – Phases - es la interfaz que muestra en que fase está el juego.
6. Dados - cuando se resuelve un combate, los dados de la izquierda (de arriba hacia abajo, ordenados de mayor a menor) pertenecen al atacante y los dados de la derecha al defensor.
7. Cards – Muestra las cartas que posee el jugador cuyo turno se esta llevando a cabo. Existen 4 tipos de cartas: planeta, galaxia, nave y comodín. Cada una con una imagen representativa y el comodín con las tres imágenes juntas.
8. Planeta – Información sobre el planeta como nombre del mismo, jugador que lo controla y naves que posee el jugador ahí.
9. Planet Links – las líneas determinan los planetas “limítrofes” para las acciones de *Attack* y *Regroup*. Los planetas que estén unidos por líneas del mismo color representan una galaxia.
10. Galaxy Links – las líneas de color *rojo* determinan las galaxias “limítrofes” para las acciones de *Attack* y *Regroup*.

6.4. Reglas

Cada jugador tiene cuatro fases de juego que representan las acciones que pueden tomar, a saber: Deploy, Attack, Draw y Regroup. Se puede saber en que fase se encuentra el jugador por la barra superior de la GUI (Graphics User Interface) y por el puntero del Mouse. Una vez que el jugador desee terminar con la fase actual debe hacer clic izquierdo sobre el botón de “Skip” ubicado en el margen superior derecho de la GUI.

Parte del GUI superior mostrando las fases

- **Deploy:** en esta fase de despliegue, el jugador puede ubicar sus fichas o naves espaciales en los planetas que posee. Si el juego recién comienza tendrá una fase de Deploy inicial para ubicar 8 naves y luego para ubicar 3 más. A partir del primer turno real del juego el jugador al comienzo de su turno recibirá un número de naves igual a la mitad (redondeando hacia abajo) de planetas que posea. Para ubicar naves el jugador deberá hacer clic derecho sobre un planeta que controle. Durante esta etapa el puntero del Mouse tendrá un símbolo “+” indicando que la acción con el botón derecho representa “agregar” (…naves).

Deploy

- **Attack:** en la fase de ataque el jugador puede tratar de conquistar planetas enemigos. Para ello debe seleccionar un planeta propio. Cuando se selecciona un planeta, con un clic izquierdo, el juego lo ayuda a distinguir los planetas “limítrofes” indicando también el color del jugador que lo posee. Con el planeta propio seleccionado se debe hacer clic derecho sobre un planeta enemigo para comenzar el ataque. El ataque en sí sigue las mismas reglas que el TEG original basado en el azar de los dados. El jugador atacante jugará con un dado menos de la cantidad de naves que posea pero nunca con más de tres. El jugador se defenderá con todas sus naves pero también con no más de tres. Al momento del ataque se tirarán la cantidad de dados correspondientes para atacante y defensor, se ubican aromáticamente de mayor a menor y se comparan uno a uno. El dado de mayor número ganará, en caso de empate la victoria se le dará al defensor. Cada dado perdedor representa una nave menos. Si el planeta defensor se quedase sin naves, el atacante pasa automáticamente una nave y tiene la posibilidad de pasar hasta tres más para continuar su ataque desde el nuevo planeta. Durante esta etapa el puntero del Mouse tendrá un símbolo de “mira” indicando que la acción con el botón derecho representa “atacar”.

Attack

- **Draw:** de la misma forma que en el TEG original se pueden robar cartas luego de la etapa de ataque, aquí sucede lo mismo. La recompensa de la carta cuando se ha logrado conquistar un país en el TEG no es automática y debe ser pedida por el jugador que en caso de olvidarse perderá ese premio (algo que sucede bastante en la realidad). Para lograr esa acción “manual”, el TEGE tampoco le da automáticamente una carta al jugador sino que en esta fase es el jugador que haciendo clic derecho sobre el botón “Draw” robará una carta. De esta forma existe la posibilidad de que el jugador se olvide y pase esta fase sin reclamar su premio agregándole un poco de esa picardía que tiene el juego original. Como se explico el botón sólo podrá ser activado si el jugador se encuentra en esa fase, si tuvo al menos una conquista y sólo una vez por turno. Durante esta etapa el puntero del Mouse no tendrá un símbolo especial indicando que no hay ninguna acción posible con el botón derecho del Mouse. El puntero será el estándar que se puede ver durante la navegación por el menú.

Estándar / Draw

- **Regroup:** antes del fin del turno el jugador puede mover las naves entre los planetas “limítrofes” que posea. Durante esta etapa el puntero del Mouse tendrá un símbolo de “flecha” indicando que la acción con el botón derecho representa “mover”.

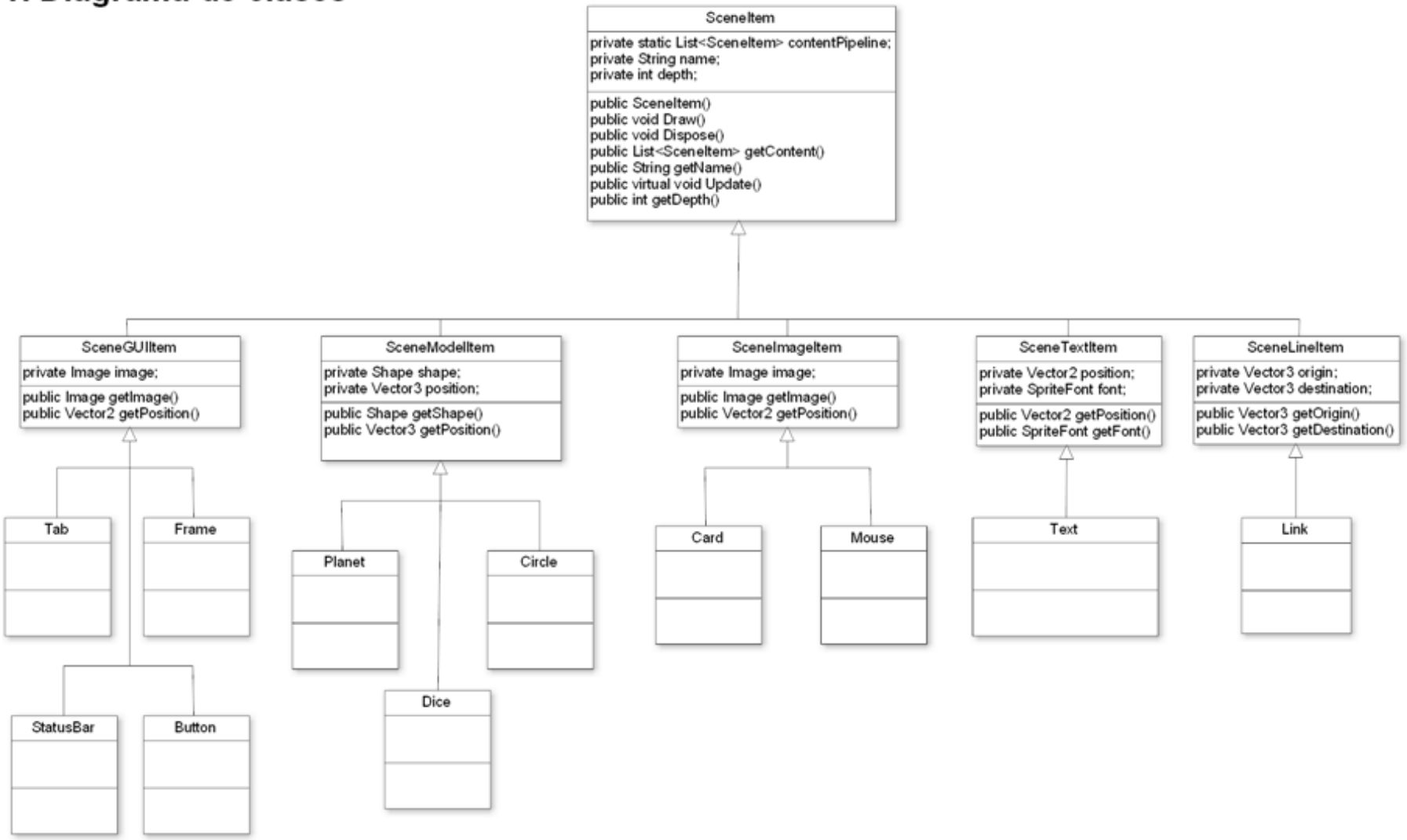
Regroup

6.5. Diseño del sistema

A continuación muestro el diseño del sistema a través del Diagrama de Clases. Para simplificarle el diagrama al lector y no sobrecargar un sólo diagrama de información es que he divido el diagrama en dos:

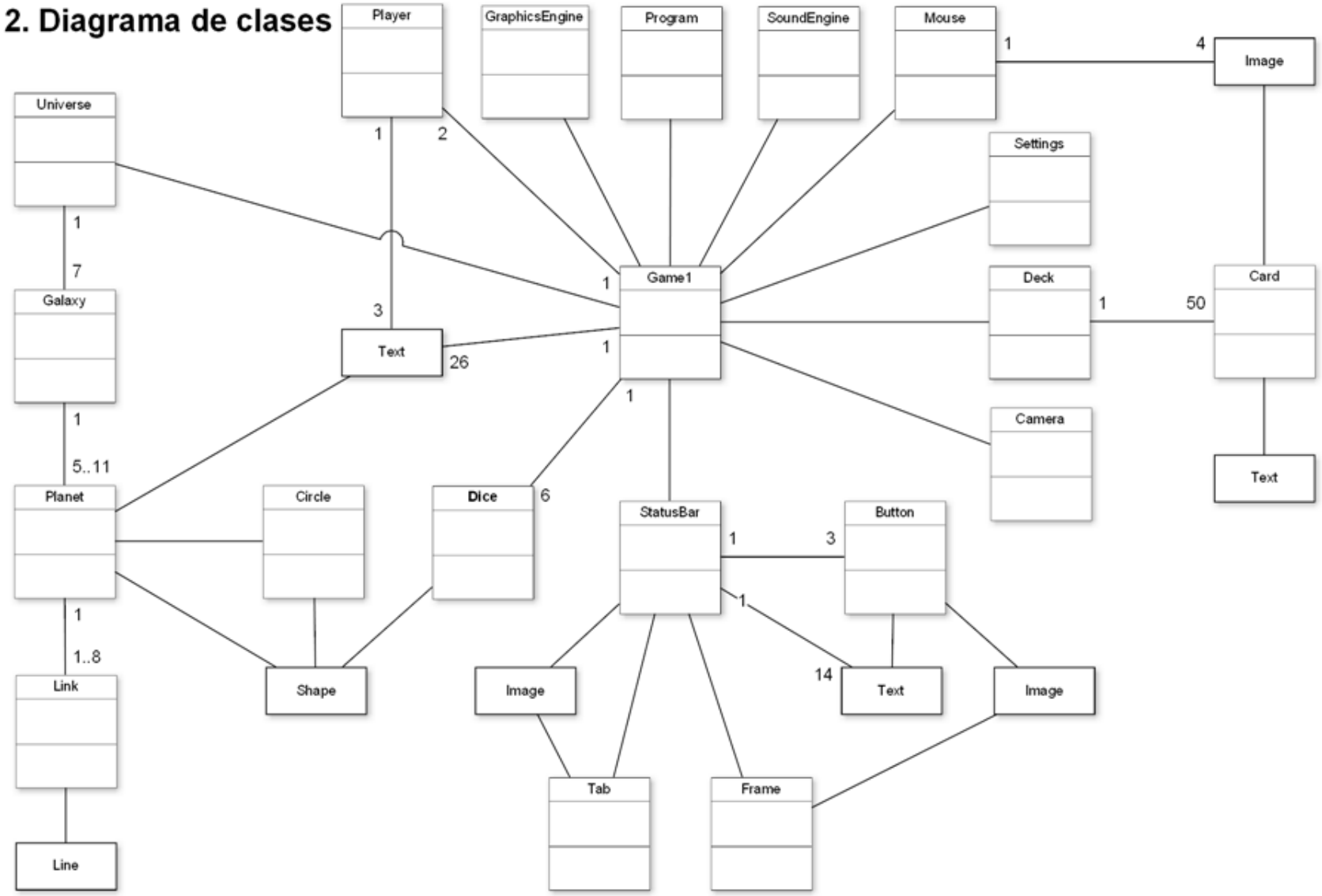
1. El primer diagrama muestra la relación de las clases en cuanto a su herencia. Este diagrama en particular es importante porque demuestra la estructura de los *SceneItems* y subcategorías para crear el *Content Pipeline* que he usado. Se han puesto sólo algunos atributos y métodos para no extender el diagrama demasiado y sólo darle una idea al lector de cuales (atributos y métodos) son mínimamente necesarios para el *Content Pipeline* y para los distintos elementos del juego. Como se puede ver, el *Content Pipeline* es una lista del la clase *SceneItem*, un atributo que es obviamente estático para que cada *SceneItem* tenga acceso pero además para que se encuentre sólo *una* instancia del mismo (un atributo estático comparte su área de memoria con todos los objetos creado de la clase donde se encuentra). El atributo que es *private* tiene obviamente su método *Getter* correspondiente. Los métodos *Draw()* y *Dispose()* no contienen lógica de dibujar o borrar propiamente dicha sino que lo que hacen es agregar el *SceneItem* al *Content Pipeline* o sacarlo del mismo. En el caso del *Draw()* se hace una validación previa para evitar agregar un objeto que ya existe en el *Content Pipeline*. Como parte de mi estructura agregue un atributo *Depth*. Este atributo indica la "profundidad" del dibujo. Para evitar estar pendiente de la secuencia en que se hace el dibujado en pantalla y evitar que por ejemplo todo quede tapado por el fondo de estrellas o que el puntero del Mouse se vea "por atrás" de los planetas, esta variable *Depth* es usada por en Engine Gráfico para darle a los objeto una secuencia en cuanto al dibujado, un valor cercano a 0 indica más próximo a la pantalla o en realidad que ese objeto debe ser dibujado por arriba de todos los demás (último) mientras que a valores más altos lo contrario. Para dar un ejemplo, el menor valor que uso es 40 y esta asignado al puntero del Mouse ya que no quiero que sea "tapado" por ningún otro objeto. El método *Update()* que será ejecutado antes de la renderización esta implementado desde la clase *SceneItem()* para ejecutarse en todo los objetos, aunque se encuentra vacío de código ya que no es realmente necesario para todo ellos. Por eso a su vez es *Virtual*, para aquellos objetos que sí usen este método (como los Planetas para su rotación) pueden hacer lo que en C# se llama *Override* del método para redefinirlo con la lógica que uno crea conveniente.

1. Diagrama de clases



2. El segundo diagrama muestra la relación de todas las clases entre sí, se eliminaron las clases *SceneItem* y subclases para evitar redundancia en el diagrama. También se omitió aclarar las relaciones “1 a 1” entre las clases para evitar llenar de las mismas el diagrama. Estas relaciones igualmente son fáciles de visualizar, se sobreentiende por ejemplo que el juego va a tener sólo un objeto GraphicsEngine, un objeto SoundEngine, un Mouse, etc. Aquellas clases representadas por un rectángulo simple indican clases creadas para representar tipos ya existentes, a saber: las imágenes que son un atributo del tipo Texture2D están ahora comprendidas dentro de la clase *Image*, los atributos Model, están representados por la clase *Shape*, el atributo String, esta dentro de la clase *Text* (que a su vez sirve para darle un enfoque más grafico que un String en sí no posee) y la primitiva LineStrip esta incluida dentro de la clase *Line*. Los juegos creados con XNA tienen automáticamente una clase *Program* que en la mayoría de los casos no se modifica, que crea una instancia de un objeto Game1 (cuyo nombre es modificable pero que decidí no cambiarlo ya que me era indiferente) y lo ejecuta. A partir de allí, como se puede observar la mayoría de los objetos van a estar relacionados con la clase Game1. La clase Camera es una clase estática ya que sólo puede haber una (en este caso del TEGE, puede ser necesario dos *ViewPorts* para ver escenas de los lugares distintos al mismo tiempo, lo que se denomina comúnmente a “pantalla dividida”). Posee los datos de su posición, define márgenes y limites y velocidades de paneo y scroll, algunos de estos datos son utilizados por la clase GraphicsEngine para crear las matrices de *View* y *Projection* correspondientes. La clase *Settings* también es estática, ya que no era necesario instanciar un objeto y quería que fuese de fácil acceso fácil para las demás clases. Posee atributos únicamente, no métodos con valores que determinan por ejemplo la profundidad de los objetos, constantes definidas, opciones de pantalla como su ancho y alto y hasta los nombre de los planetas. La parte de los Settings (configuraciones) siempre esta presente, a veces en forma de archivo de texto plano o como preferí en este caso dentro de una clase particular para evitar que las opciones puedan ser modificadas.

2. Diagrama de clases



6.6. Diseño Gráfico

6.6.1. 2D

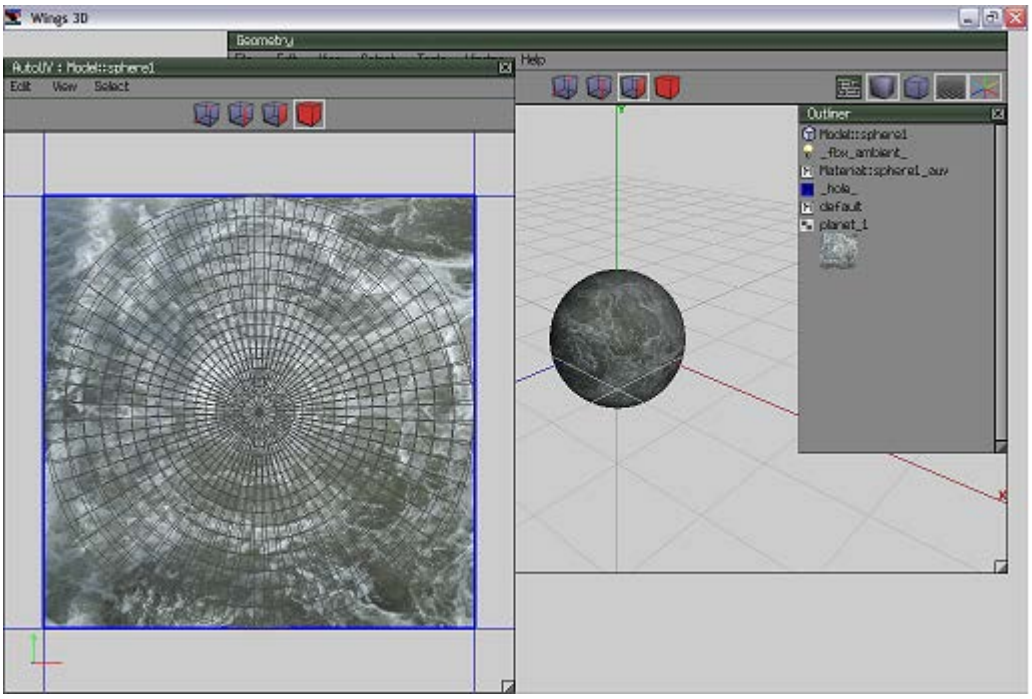
Por falta de habilidades personales en el dibujo técnico, solicite ayuda a la hermana de un amigo mío que sí posee talento artístico y usa herramientas de diseño gráfico. Al consultar con el Director de Carrera se concluyó que la carrera de Ingeniería Informática no estaba apuntada a lo artístico en el dibujo por lo que la evaluación no recae sobre este punto y la ayuda externa es permitida en este caso. Carolina Rocco estuvo a cargo de la GUI principal del juego, marco general, botones, fuentes y cartas. Yo hice prototipos para saber el tamaño de las imágenes para empezar a manipularlas en código, esos prototipos se los pase a ella junto a algunos comentarios generales para que sepa justamente el tamaño y lo que necesitaba. A medida que los iba terminando me los pasaba y actualizaba las imágenes sin necesidad de modificar el código. Pero para igualmente tener conocimiento sobre esta área de los videojuegos también le pedí que me cuente sobre su desarrollo en este proyecto desde su posición y conocimientos de la misma forma que lo estoy haciendo yo en los otros elementos. Cito sus palabras textuales:

"La selección de colores son todos metálicos, relacionado con lo que es espacial o relacionado con naves y tableros que utilizaban colores fríos, azules y grises. En base al pedido de un tablero con motivo espacial se hizo una búsqueda de los cuales se basó el diseño. Las teclas son de aviones pero con los efectos se le dio un aspecto más moderno.

El programa de diseño utilizado fue el Fireworks de Macromedia. Se uso este programa por las facilidades de efectos y sombras para este tipo de tareas. Usar algún programa de manejo vectorial no aplicaba porque se usaron fragmentos de imágenes y además por las facilidades mencionadas anteriormente, sumada también a la facilidad personal con el programa por tener experiencia con el mismo." Carolina Rocco.

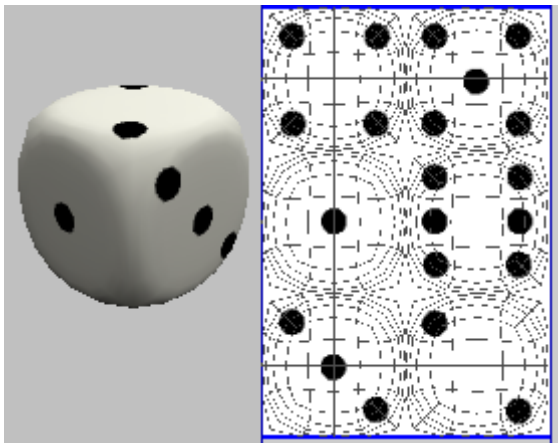
6.6.2. 3D

Los elemento 3D que se pueden observar en el juego (Planetas, Aro y Dados) los hice con el programa Wings3D, que es gratuito y permite comercializar libremente todo los modelos que hagamos con él sin restricciones. Los planetas y dados poseen una textura aplica que a mi juicio creaban una efecto de un planeta real, repito que el dibujo en general no es mi fuerte. Las texturas o imágenes para los planetas fueron sacadas de Internet de páginas que ofrecen texturas gratis de cualquier tipo, y puse más énfasis en los planetas pertenecientes a nuestra vía Láctea ya que saque esas texturas de la página de la NASA especialmente. Un planeta es un Mesh único con una textura aplicada. El Mesh en dividido en segmento que luego van a ser ubicados arriba de la textura indicando que posición del modelo y que parte de la textura queremos asociar.

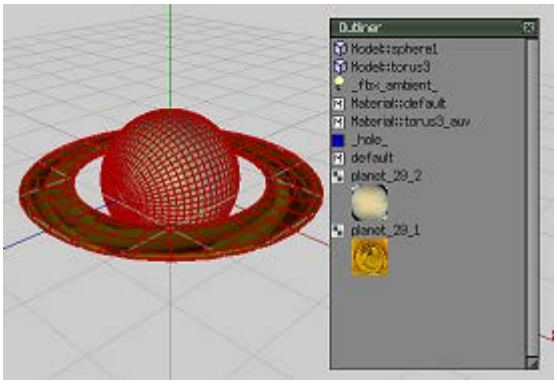


Wings 3D

Dentro de los planetas hay dos que estan copuestos por más de un mesh, ellos son los que poseen un anillo como Saturno y Urano. Cada Mesh tiene su propia textura. En el caso de los dados, la subdivisión de los segmentos se hizo sobre cada cara y se ubica a cada cara sobre una parte de una textura que cree con los puntos negros que representan los números. El Aro que envuelve a los planetas cuando se los selecciona no tiene ni un color ni una textura aplicada, eso me permite en código cambiarle el color al que posee el jugador y representar mejor así su posesión.



Dado con su textura



Planeta y Aro juntos

6.7. Engine Gráfico y Content Pipeline

Como ya se vio decidí crear mi propia estructura que soporte el concepto de *Content Pipeline*. Esta lista irá siendo modificada a lo largo de la ejecución y pasada al Engine Grafico al finalizar cada Frame para su renderización. La clase la definí como *SceneItem* para respetar terminología y cada elemento tiene acceso a la lista para poder agregarse o quitarse de la misma. Las subclases fueron implementadas para categorizar todos los elementos que difieren en su renderización, como texturas, modelos, la primitiva de línea y todo lo que sea String que tiene que ser visto en pantalla. Creé una clase *SceneGUIItem* también, que en estructura es igual a la *SceneImageItem* ya que ambas manejan texturas pero las dividí por un sentido de ordenamiento y por si acaso en el futuro encontraba alguna diferencia en particular, ya las tenía separadas. El Engine Gráfico lo que hace es dependiendo de cada subclase aplica el método *Draw()* correspondiente como se explico anteriormente. En Engine Grafico no aplica ningún algoritmo de *culling* y/o *clipping* ya que no se justificaba ese procesamiento extra para tan pocos elementos en pantalla, que en algunos casos hasta se pueden ver todos ellos. *Culling* y *clipping* justifican su algoritmo cuando eliminan miles de objetos y realmente reducen la carga a la hora de renderizar, en este caso semejante procesamiento devolvería resultados similares a los que ya se ven sin procesar.

6.8. Engine de Sonido

Por el estilo de juego los sonido no se escuchan en cualquier tiempo sino que sólo cuando el jugador realiza una acción particular. Por esto y por estar mirando siempre hacia delante, hacia un tablero, no encontré necesario aplicar un sistema de sonido más allá del sistema Mono. Al no poseer gran cantidad de sonidos y para evitar agregar más información en varias clases decidí poner todas las Cue's de sonido en el Engine, para que las cargue y las reproduzca sin necesidad de interactuar con otros objetos. Todas las modificaciones posibles quedan centralizadas en esta Clase. Por las opciones que agregué los sonidos tienen dos categorías posibles "Music" o "Default". De esta forma en vez de tener un control de volumen generalizado puedo modificar los valores para lo que es música y sfx por separadp, ya que utilizan distintos parámetros de volumen. Para la creación de las pistas de Audio utilice el Microsoft Cross-Platform Audio Creation Tool (XACT) que viene con el XNA Framework.

6.9. Métricas TEGE

Se consideró interesante aplicar la misma prueba de FPS al proyecto en sí para determinar su performance, para tener una idea de cómo se comporta una aplicación más compleja que no son solamente cuadrados rebotando contra los márgenes y para darle al lector un panorama del tamaño que tiene el proyecto. Para este caso el número entre

paréntesis indica que en el juego se pueden encontrar en pantalla de 147 a 170 elementos al mismo tiempo.

Estas pruebas se corrieron en una PC con Windows XP, Intel Core2 Duo 2,4GHz, con 2GB de memoria RAM y placa de video NVIDIA GeForce 7950GT de 512MB de memoria.

	TEGE		
Líneas de Código	4768		
Cantidad de Archivos	Button.cs Camera.cs Card.cs Circle.cs Deck.cs Dice.cs Frame.cs Galaxy.cs Game1.cs GraphicsEngine.cs Image.cs Line.cs	Link.cs Mouse.cs Planet.cs Player.cs Program.cs SceneTextItem.cs SceneGUIItem.cs SceneImageItem.cs SceneItem.cs SceneLineItem.cs SceneModelItem.cs Settings.cs	Shape.cs SoundEngine.cs StatusBar.cs Tab.cs Text.cs Universe.cs TOTAL: 30
Horas/hombre	5 meses - 4-6hs por día		
Tamaño del proyecto	Audio - 3 – 44,9MB Fonts - 3 – 0,15MB Images - 19 – 4,56MB Models - 103 – 6,58MB Textures - 6 – 16,8MB TEGE.exe – 0,08MB TOTAL: 73,07MB		
Memoria en uso (147 - 170)	102.444 KB		
Uso de CPU (147 - 170)	10 - 15%		
FPS (147 - 170)	60		

7. Conclusiones

El objetivo se ha cumplido con creces. Mi iniciación en el mundo 3D fue simple pero enriquecedora, lo próximo a ver y utilizar serán las animaciones. La creación de mi propio *Content Pipeline* y *Engine Gráfico* me permitió entender mucho más del tema que pensaba antes de comenzar este proyecto que era muy complejo y que lo utilizaban sólo grande compañías desarrolladoras. Aún más, este enfoque me permitió darme cuenta que es posible embarcarse en proyectos aún más grandes sabiendo que los límites sólo se los pone uno. Desarrolle un juego con su lógica completa, gran calidad gráfica y distribuible. Este videojuego sacando algunos ítems del contenido como la música y cambiando las fuentes, ambas cosas con autores propios puede ser comercializado sin problemas e incluso, modificando algunas líneas de código, distribuido por el servicio Xbox Live. Siempre tuve entendido que el sonido era lo último en hacerse del proyecto y entendí que es porque es lo más fácil de hacer y de acoplar. Quedará como cambios futuros portar el juego al XNA Framework 2.0 y/o 3.0 para agregarle funcionalidad multi-jugador e Inteligencia Artificial (jugadores controlados por la máquina).

Bibliografía

URL's

- http://www.riemers.net/index.php
- http://www.answers.com/topic/ray-tracing
- http://www.ziggyware.com/forum/viewthread.php?forum_id=12&thread_id=13321&pid=44161
- http://www.wikipedia.org/
- http://www.programacion.com/java/tutorial/ags_j2me/1/
- http://www.developer.com/java/article.php/893471
- http://www.sdltutorials.com/sdl-tutorial-basics/
- http://www.vgchartz.com/
- http://msdn.microsoft.com/en-us/library/aa446533.aspx
- http://www.theesa.com/
- http://www.mpaa.org/

Libros

- "3D Game Engine Design", Autor: David H. Eberly.
- "Introduction to 3D Game Engine Design Using DirectX 9 and C#", Autor: Lynn Thomas Harrison.
- "Tricks of the 3D Game Programming Gurus", Autor: André LaMothe.

Documentales

"Rise of the videogame", Discovery Channel

Herramientas

Microsoft Visual C# IDE

Microsoft XNA Framework

CodeBlocks (C/C++) IDE

Librería SDL (Simple Direct Layer)

Java Sun NetBeans IDE

Wings3D

Pacestar UML Diagrammer

Macromedia Fireworks

GIMP